Roma Tre University
Ph.D. in Computer Science and Engineering

Information Management in the Distributed Web

Enrico Marino

Information Management
in the Distributed Web

A thesis presented by

Enrico Marino

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in Computer Science and Engineering

Roma Tre University
Dept. of Informatics and Automation

2018

ADVISORS

Prof. Alberto Paoluzzi
David Dias

REVIEWERS

Prof. Ernest Cachia
Prof. Miguel-Angel Sicilia Urban

Do nothing from selfish ambition or conceit,
but in humility value others above yourselves.
Let each of you look not to your own interests,
but to the interests of others.
—Philippians 2:3-4

TABLE OF CONTENT

The World Wide Web 22
The content-addressed peer-to-peer networks 23
Web Content Management Systems 24
Conclusions 25
Abstract 26
Why the title 26

CHAPTER 1
THE WORLD WIDE WEB 27
1.1 The origin of the Web 28
1.1.1 The basic idea: A distributed HyperText 28
Why the name HyperText 28
The origin of the HyperText 29
1.1.2 The first Web 30
1.1.3 “This is for everyone” 31
1.1.4 The Web and Internet 32
The Internet 32
The Web 33
1.2 The protocols of the Web 34
1.2.1 The Internet Protocol suite 34
1.2.2 The HyperText Transfer Protocol (HTTP) 35
1.3 The languages of the Web 36
1.3.1 The Hypertext Markup Language (HTML) 36
Syntax 36
Structure 37
Representation 37
Semantic 38
1.3.2 The Cascading Style Sheet (CSS) 39
Separation of content and presentation 39
The loose error-handling 40
Declarative languages 41
1.3.3 JavaScript 42
Why the name JavaScript 42
The use of JavaScript 43
The DOM 43
The Asynchronous JavaScript and XML (AJAX) 44
1.3.4 The Extensible Markup Language (XML) 45
1.3.5 The JavaScript Object Notation (JSON) 46
Data Types 46
Number 46
String 46

Boolean 47
Null 47
Array 47
Object 47
1.4 The evolution of the Web 48
1.4.1 The static Web (the Web 1.0) 48
1.4.2 The dynamic Web (the Web 2.0) 49
1.4.3 The social Web 50
1.5 The Semantic Web 51
1.5.1 Microdata 52
Itemscope and itemtype 52
1.5.2 Schema.org 53
Types and properties 54
1.5.3 Open Graph Protocol 55
Basic Metadata 55
Optional Metadata 56
Structured metadata 56
Types 56
1.5.4 AMP 57
The AMP HTML format 58
1.6 The problems of the Web 59
1.6.1 The Inefficient Web 60
1.6.2 The Lost Web 61
1.6.3 The censored Web 62
1.6.4 The Offline Web 63
1.6.5 The Centralized Web 64
7.3 The attempts to re-decentralize the Web 65
1.6.6 The Advertising Web 66
1.6.7 The Tracking Web 67
1.6.8 The Fake Web 68

CHAPTER 2
THE CONTENT-ADDRESSED
PEER-TO-PEER NETWORKS **71**
2.1 The Peer-To-Peer Networks 72
2.1.1 Architecture 72
2.1.2 File-sharing applications 72
Napster 72
BitTorrent 73
2.1.3 Content-addressing 74
The implications of content-addressing 75
Content-addressing increases the durability of data 75
Content-addressing increases the integrity of data 75
Content-addressing allow content to be unavailable for a period 75
Content-addressing is harder to attack and easier to recover 75

2.2 The InterPlanetary File System 76
2.2.1 IPFS network 77
2.2.2 IPFS Objects 78
File System 78
2.2.3 The InterPlanetary Linked Data 79
Merkle links 80
Pesudo-link objects 81
Merkle DAG 82
Merkle paths 82
Data model 83
2.2.4 The InterPlanetary Naming System 84
Human-readable mutable addressing 84
The IPFS/HTTP gateway 84
2.3 Dat 85
2.3.1 Beaker Browser 86
Website manifest 86
DatArchive API 87
WebDB 88
2.3.2 HashBase 89
2.4 ZeroNet 90
2.4.1 Using ZeroNet 90
Create a site 90
Visit a site 90
Update a site 91
Update a site of another user 91
2.5 Modern Cryptography 92
2.5.1 Private and Public Keys 92
Private Keys 92
Public Keys 93
2.5.2 Key Formats 93
Deterministic Key generation 93

CHAPTER 3
WEBSITE DESIGN AND
THE CONTENT MANAGEMENT SYSTEMS **95**
3.1 Websites Design 96
3.1.1 Static Websites 96
Website pages 97
Why the name Blog 97
Index page 98
Single pages 99
3.1.2 Dynamic websites 100
PHP 100
Why the name PHP 100
Brief history of PHP 101

Using PHP 101
Index page 102
Single page 103
Routing 103
3.2 The Content Management Systems 104
3.2.1 WordPress 105
How WordPress works 105
The Architecture of WordPress 106
WordPress ecosystem 106
Plugins marketplace 107
Themes marketplace 107
3.2.2 Templates 108
Index template 108
Archive template 109
Category template 109
Template Tags 110
Template partials 110
Include Tags 110
Template Hierarchy 111
3.2.3 Post Types 112
Posts 112
Pages 112
Attachments 112
Custom Post Types 113
3.2.4 Plugins 114
Plugin repositories 114
Akismet 114
Hello Dolly 114
Plugin Hooks 114
Actions and Filters 115
3.2.5 Themes 116
Required files 116
Common template files 117
3.3 Static Site Generators 119
Hosting static websites 119
3.3.1 Content 120
Markdown 120
Shortcodes 122
Frontmatter 123
3.3.2 Site structure 124
Page Bundles 125
3.3.3 Content types 126
Archetypes 127
3.3.4 Taxonomies 128

3.3.5 Themes 129
Layout templates 130
Partial templates 130
Static files 130
3.4 Decoupled Web Content Management Systems 131
3.4.1 Decoupled Web CMS 131
Tightly-coupled vs loosely-coupled systems 131
3.4.2 Headless CMS 132
3.5 Static Progressive Web App Generators 133
3.5.1 Gatsby.js 133
PRPL Pattern 133
Directory structure 134
3.5.2 Components 135
Page Component 135
Page template components 136
Layout components 137
HTML component 138
3.5.3 Data Management 139
Data Nodes 139
Querying Data Nodes with GraphQL 141
Gatsby's GraphQL schema 143
3.5.4 Plugins 144
The Bootstrap processes 144
Source plugins 145
Transformer plugins 145
3.5.5 Building a Blog site 146
Read and transform markdown files 146
Posts page 147
Post page 147
Tags page 148
Tag page 149
Create pages 151

CHAPTER 4
FRONT-END WEB FRAMEWORKS **153**
4.1 Single Page Applications 154
Frameworks vs. Libraries 155
4.1.1 MVC Front-end Web Frameworks 156
MVC pattern 156
Observer pattern 158
4.1.2 Templating System 160
4.1.3 Dynamic rendering 161
Data binding 161
Virtual DOM 161

4.2 Angular 162
    4.2.1 Modules 162
    4.2.2 Components 163
    4.2.3 Templates 164
        Template syntax 164
    4.2.4 Data binding 165
        Text interpolation 165
        Property binding 165
        Event handling 165
        Two-way binding 165
4.3 Vue.js 165
    4.3.1 Data binding 166
        Binding inner text 166
        Binding attributes 166
        Conditional rendering 167
        Repeat rendering 167
4.4 React 168
    4.4.1 Elements 168
    4.4.2 JSX 169
        Updating attributes and properties 171
        Repeat rendering 171
        Handling events 172
    4.4.3 Components 173
        Defining Components 173
        Composing Components 174
        Component props 175
        Component state 175
    4.4.4 The data flow 176
    4.4.5 The Virtual DOM 177
        Reconciliation 177
4.5 Redux 177
    4.5.1 Principles 179
        Single source of truth 179
        State is read-only 179
        Immutability 179
        Changes are made with pure functions 179
    4.5.2 Concepts 180
        Actions 180
        Action creators 180
        Reducers 180
        Store 181
4.6 Web Components 183
    4.6.1 Custom Elements 184
        Defining a custom element 184

Custom Element name 184
Custom Element constructor 185
Using a custom element 185
Custom element reactions 186
Custom elements content 187
4.6.2 Templates 188
4.6.3 Shadow DOM 189
4.7 CSS frameworks 190
Restyleable HTML 190
Reusable CSS 191
4.7.1 Bootstrap 192
Grid system 192
Containers 192
Rows 192
Columns 192
Responsive breakpoint 194
Alignment 195
Vertical alignment 196
Horizontal alignment 196
Reordering 196
Utilities 197
Border 198
Colors 198
Display 198
Components 198
Alerts 199
Badges 199
Breadcrumb 199
Buttons 199
Card 200
Nav 200
Forms 200
List group 201
Pagination 201
Dynamic components 202
4.7.2 Atomic CSS 203
Display 203
Dimensions 203
Border-box 203
Widths 203
Max widths 204
Heights 204
Spacing 204
Margin and padding 206

Flexbox 206
Flex 207
Align items 207
Justify content 207
4.7.3 Maintainable CSS 208
Semantics 208
Semantic classes are readable 209
Semantic class ease building responsive sites 209
Semantic class are easier to find and to debug 210
Semantic class eliminate the risk of regression 210
Semantic class provide hooks for automated tests and JavaScript 210
Semantic class don’t need maintaining 210
Semantic class are recommended by the standards 210
Reuse 210
Conventions 211
Modules 212
State 213
Modifiers 214
4.7.4 Styled Components 216
Interpolation 217
Styling any components 218
Extending Styles 219
4.7.5 Styling Web Components 220
:shadow and /deep/ 220
Custom properties 220
@apply rule 221
::part and ::theme pseudo-elements 222
::part pseudo-element 222
Forwarding parts 223
Rename forwarding 224
Prefix forwarding 224
::theme pseudo-element 226

CHAPTER 5
DESIDÈRA
THE CONTENT MANAGEMENT SYSTEM
FOR THE DECENTRALIZED WEB 229
5.1 Motivation 230
Web Content Management Systems 230
Static Site Generators 231
Decoupled and Headless Web CMSs 232
Desidera 232
Why the name 233
5.2 Principles 234
Design content at first 234

Design content by data 235
5.3 Data 236
5.3.1 Objects 236
5.3.2 Paths 236
5.3.3 API 236
5.4 Types 237
5.4.1 Content types 238
Properties 238
Controls 238
Applicable widgets per field type 239
Widget settings 241
5.5 Content 242
5.5.1 Organizing content 242
5.5.2 Page type 243
5.5.3 Resource type 244
5.5.4 Post type 244
5.6 Theme 244
5.6.1 Components 245
Page components 246
Part components 246
5.7 The App 247
5.7.1 The main files 247
index.js 247
index.html 247
5.7.2 Routing 248
Tree of routes 248
Route transition 249
5.7.3 The main process 250
5.8 The workflow 251
5.8.1 Updating a website 251
5.8.2 Publishing a website 252
5.8.3 Linking a website 253

CHAPTER 6
PANTAREI
THE RESILIENT FRONT-END WEB FRAMEWORK
BASED ON WEB COMPONENTS **255**
6.1 Motivation 256
6.2 Principles 258
Why the name Pantarei 258
6.3 Directives 259
6.3.1 Syntax 260
6.3.2 The Element constructor 261
6.3.3 Built-in directives 262
6.3.4 Update an attribute 262

6.3.5 Update a property 263
HTML Attributes vs. DOM properties 264
6.3.6 Update the class list 265
6.3.7 Update the inner text 266
6.3.8 Repeat rendering 267
Nested repeat 268
6.3.9 Conditional rendering 269
6.3.10 Handling Events 270
6.4 Custom directives 271
6.4.1 Create a custom directive 271
Install a new directive 271
6.4.2 Extend built-in or custom directives 272
6.4.3 Directives mapping 273
6.5 Components 274
6.5.1 Component definition 274
6.5.2 Component properties 276
Properties validation 277
6.5.3 Composition 278
6.5.4 Distribution 279
6.6 Designing a website 280
6.6.1 Content types 280
6.6.2 Content models 282
Routes 283
6.6.3 Theme 284
Page Posts 286
Page Post 286
Page Tags 288
Page Tag 289
Store 289

CHAPTER 7
CONCLUSIONS **291**
7.1 The state of the art 291
7.2 Pantarei 292
Learning Curve 292
JavaScript idioms/extensions 292
Component ecosystem 293
Optimizations 293
7.3 Desidera 294
7.3.1 Content Delivery 294
Delivery over a peer-to-peer network 294
Delivery over a CDN 294
7.3.2 Dev Op Experience 294
Serverless 294

Hosted 294
7.3.3 User Experience 295
Offline access 295
Prefetch linked pages 295
Page caching 295
7.3.4 Front-end Developer Experience 296
Componentization 296
Declarative component 296
Unidirectional data flows 296
7.4 Future works 297
Pantarei 297
Desidera 297
7.4.1 Enquire - the Search Engine for the Distributed Web 298
YaCy 298
7.4.2 Enquire 299
Why the name 299

PREAMBLE

There is always one motivation that excites us more than others. Elevated by aspirations and towed by suggestions, step by step, we trace our path. Being aware of what truly motivates us is key to understanding where we are going. Whatever the path may be, wherever your starting point is, the real motivation allows us to get the right direction. Because, the important part of a journey is not the journey itself, neither the destination, but the motivation.

> *If a man does not know to what port he is steering,*
> *no wind is favourable to him.*
> —Seneca

Sometimes a goal is not completely defined. No one would start walking if they didn't define a goal, no one could reach a goal if they didn't start walking. As you move ahead the goal becomes clearer, no matter how long the distance. Taking the smallest possible steps, you can accomplish the most unimaginable goal.

> *Do what is necessary, then what is possible.*
> *And suddenly you'll be surprised to do the impossible.*
> —Saint Francis of Assisi

Small steps, giants leaps.

> *One small step for a man,*
> *one giant leap for mankind.*
> —*Neil Armstrong*, from the moon

1969. The croaking sound of the voice on the radio still echoes in our ears.
A man - Neil Armstrong - takes the first step on the moon[1]. The interplanetary voyage became a reality.

In that same year—on earth—something was about to change the way we would communicate —to shorten any distance and allow us to be united.

[1] Natalie Wolchover. (August 2012). *"One Small Step for Man": Was Neil Armstrong Misquoted?* (https://goo.gl/weaEGn)

1969. Two separated university computer networks are about to be united for the first time —the precursor of Internet, the network of networks, was rising[2].

No breakthrough can take place unless there has been much preparation work before it. The history of human civilization is a tale of cumulative effort. Each generation builds upon the work of their forebearer. Step by step our species makes progress. Whether the progress is incremental or a huge leap forward, it is always borne upon the accomplishments of those who came before.

Nowhere is this layered progress more apparent than in the history of technology. Even the most dramatic endeavors in technological advancement are only possible when there is some groundwork to build upon.

Technologies aren't created in isolation.
The printing press, for example, would not have been invented by Gutenberg, if someone, before him, had not created the screw press to make wine.
Typewriters would not have been invented, if Gutemberg, in turn, had not introduced the replaceable, moveable letters for his printing press.

Technologies are imprinted with the ghosts of their past.
The layout of the "qwerty" keyboard for personal computer, for example, is an echo of the design of the first generation of typewriters. That specific arrangement of keys was chosen to reduce the chances of mechanical pieces of metal clashing as they sprang forward to leave their mark on the paper.

Scientific progress would be impossible without a shared history of learning and experiences to draw from. Sharing knowledge has always been the key to new and open perspectives.

> *If I have seen further,*
> *it is by standing on the shoulders of giants.*
> —Isaac Newton

Our history of sharing knowledge has been a long relay race, where the baton is the knowledge and its flame is a wisdom to be nourished. A race signed by obstacles and barriers along the way. However, no fence can hinder the desire to share and to come together.

1989. The Berlin wall, the symbol of the division from a global war, falls.

In that same year, something was about to change the way in which we would interact, breaking through every known barrier.

---

[2] G. Gromov (2012). *Roads and Crossroads of the Internet History*. (https://goo.gl/PnG6xJ)

1989. Among the mountains on the border between Switzerland and France, in the laboratories at CERN, a computer scientist from England was tackling the thorny problem of information management on a large scale.

An unassuming document with the title "*Information Management: A Proposal*"[3] is the attempt to persuade the management at CERN that a unified information management system was in their best interests, in order to foster the synergy of the various research departments. Fortunately, the supervisor, Mike Sendall, recognized the potential of this idea and gave the go-ahead by scrawling the words "*vague but exciting…*" across the top of the cover and a question along the bottom: "*and now?*".

That computer scientist is Timothy John Berners-Lee and his proposal would become the World Wide Web.

[3] Tim Berners-Lee (1989). *Information Management: a proposal.* (https://goo.gl/SkB9t1)

INTRODUCTION

WWW. Just three letters contain a universe.
The World Wide Web, known as WWW or simply Web, is

> "*the universe of network-accessible information,*
> *the embodiment of human knowledge*"[4].

The Web was originally conceived and developed to meet the demand for automatic information-sharing between scientists in universities and institutes around the world[5]. The Web would then proved to be a communications tool to allow anyone, anywhere to share information.

During the years, the Web has evolved and influenced how we interact, how we conduct business, how we keep learning and receive news, in general, how we deal with daily life, and nowadays, its use is so ubiquitous that its existence is taken for granted.

At the same time, the usefulness of the Web as a foundation for the distribution and persistence of information worldwide, as the sum of human knowledge, has shown its weakness. The Web is very extensive, in continuous expansion, but also very fragile, in continuous decay.

The Web has unified the entire world into a single global information space, standardizing how we produce and present information to each other, but the way content is distributed has is fundamentally faulty.

> *We begin in admiration*
> *and end by organizing our disappointment.*
> — Gaston Bachelard

[4] W3C. (1995). *About The World Wide Web.* (https://goo.gl/kmGrMY)
[5] CERN. *The Birth of the Web.* (https://goo.gl/2UcTwW)

The World Wide Web

While the Internet is a truly distributed system, designed so that if any one piece goes out, it will still function —the Web, built on top of the Internet, is not. The problems is in the way it identifies and distributes content: by means location. The location-addressing approach of the Web has led to a series of serious consequences, one linked to the other.

The Web is unreliable. If a content is moved to another location, or just renamed, any link to that content - the previous location - would be no longer traversable. Also, if the server that hosts the content is turned off, or if it is made inaccessible, in any way, for whatever reason, any content hosted by that server would be no longer available.

The Web is inefficient. Even if a thousand people have downloaded a thousand copies of the same content, to a thousand different physical locations, all references to that content would still point to that original, single location.

And yet, since its development, the Web has steadily evolved into an ecosystem of large, corporate-controlled giant-platforms which intermediate information online. This has lead to problems ranging from censorship at the behest of national governments to more subtle, perhaps even unintentional, bias in the curation of content users see based on opaque, unaudited curation algorithms.

How to evelove the Web?

From one perspective, evolving the Web infrastructure is nearly impossible, given the number of backwards compatibility constraints and the number of strong parties invested in the current model. But from another perspective, new protocols have emerged and gained wide use since the emergence of the Web, having the potential to resolve most of the problems the Web suffers from.

The content-addressed peer-to-peer networks.

The content-addressed peer-to-peer networks

There have been many attempts at constructing a global peer-to-peer network. Some systems have seen significant success, and others have completely failed. However, while there have been successful repurposings, no general system has emerged that offers global, low-latency, and decentralised distribution, as infrastructure to be built upon.

Finally, from the last few years, a novel peer-to-peer, version-controlled, distributed network has been emerging and gaining wide usage: the InterPlanetary File System (IPFS)[6]. Based on a content-addressing approach, the peer-to-peer network provided by IPFS can address and resolve most of the problems the Web suffers from.

The peer-to-peer network provides just the infrastructure on which a truly distributed Web can be built. Since the Web is made by websites, building the Web implies building websites, and vice versa.

How to build websites over a content-addressed peer-to-peer network?

We should investigate which systems, and technologies and methodologies, have been developed and established to build websites for the current Web. Then we could combine these systems on the peer-to-peer infrastructure, or figure out new systems for that.

[6] Juan Benet. (2014). *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)*. (https://goo.gl/zP7RcG)

Web Content Management Systems

Since the beginning of the Web, the need to ease the publishing process made the success of the Web Content Management Systems (Web CMS): systems that allow you to create, modify, review and publish content, and so create websites, on the Web.

Actually, there are hundreds of Web CMSs that can be used[7], but very few of them have clearly become predominant over the rest of the competition. In particular, the most used one, *WordPress*, is running over one third of all the website in the Web. Given its wide usage, it is taken as a valid explanatory instance.

In general, when a user visits a website, built with a Web CMS, expecting the latest content from the server, that hosts the content, the Web CMS queries the database to get the content, passes the results to its templating engine, that compose the HTML, and serves the page.

Static site generators shift the heavy load from the moment users request the webpage to the moment the webpage actually changes (when the website is updated), generating a structure of purely static HTML files that are ready to be delivered as-are to the users.

Choosing a Web CMS means accepting not only the language it is written in, but also its editing and administration tools, its database, its templating system, etc. Decoupled Web CMSs aim to improve this situation.

A Decoupled CMS is essentially a regular full stack of content management, delivery, and presentation solution but allows for content stored within it to be leveraged by other systems.

Decoupled CMSs delegate the presentation to Single Page Applications.

However, the systems, methodologies, and technologies, already used to build websites are not thought, nor ready, to design, build and deploy websites over a peer-to-peer network. And even if they were used as-are on a peer-to-peer network, they can not make the most of such a network, as a system designed from scratch could do.

How to build websites over a content-addressed peer-to-peer network?

Enter the proposal.

[7] W3Techs. *Usage of content management systems for websites*. (https://goo.gl/6XqFst)

Desidera

The thesis introduces Desidera, a novel approach to design and deploy websites, over a global content-addressed peer-to-peer network.

Deisdera is composed by two systems: the InterPlanetary Web Space (IPWS), a versioned, data-driven content-management system, to manage and deploy content onto a content-addressed distributed file system (i.e. the InterPlanetary File System, IPFS); and Pantarei, a front-end Web framework based on HTML5 Web Components standards, to design resilient user-interfaces for websites that are delivered and distributed over a peer-to-peer network.

The main principle that drives both the systems is the separation of concerns, in every aspect of the design, to avoid as much as possible any lock-in, and to minimize the redistribution of the content and the parts of the websites.

In essence, websites are made with Web Components and delivered to the peer-to-peer network provided by the InterPlanetary File System.

Pantarei

Pantarei, is a resilient front-end Web framework to design reactive user-interfaces.

Pantarei is based on the HTML5 Web Components standards, in order to ease the process to extend and customize websites, to design components that can be generally used in the most natural and longeval way for the Web: using Web Standards.

Conclusions

The proposal enables a user-centric, resilient, permanent, versioned, decentralized Web.

The work described in this thesis aims to be a small step in the path to "*unlock the Web open*"[8] as it was originally intended to be: a universal space to allow anyone, anywhere to share information, to share knowledge — "*for everyone*"[9].

[8] Brewster Kahle. (August 2015). *Locking the Web Open: A Call for a Decentralized Web*. (https://goo.gl/fDUmgP)

[9] J. Keith (November 2013). "*This is for everyone*" (https://goo. gl/WofL58)

Abstract

The World Wide Web, or simply the Web, is very extensive, in continuous expansion, but also very fragile, in continuous decay. The Web has unified the entire world into a single global information space, but the way information is preserved and distributed is fundamentally faulty. The fundamental problem rely on the way content is identified: by means location.
A location-addressed Web means a Web where content can be lost, overwritten, censured, inefficiently distributed, and so centralized.

The thesis claims that to resolve the problems of the Web, the Web should be built over a content-addressed peer-to-peer network.

In particular, the thesis introduces Desidera, a novel approach to design and deploy websites, over a global content-addressed peer-to-peer network, and two systems: the InterPlanetary Web Space (IPWS), a versioned, data-driven content-management system, to manage and deploy content onto a content-addressed distributed file system (i.e. the InterPlanetary File System, IPFS); and Pantarei, a front-end Web framework based on HTML5 Web Components standards, to design resilient user-interfaces for websites that are delivered and distributed over a peer-to-peer network.

The proposal enables a user-centric, resilient, permanent, versioned, decentralized Web.

Why the title

The title "*Information Management in the Distributed Web*" is a reference to "*Information Management: a Proposal*" by Tim Berners Lee[10] that laid out the structure and theory of the Web as we use it now.

[10] Tim Berners Lee. (March 1989). *Information Management: a proposal.* (https://goo.gl/SkB9t1)

CHAPTER 1
THE WORLD WIDE WEB

In this chapter, the World Wide Web is explored.
How the Web was born: the basic idea that gave the Web its success.
How the Web works: the protocols that allow the Web to work.
How the Web is built: the languages that allow the Web to be built.
The evolution of the Web: the Web 1.0, the Web 2.0, and the Semantic Web.
Finally, the problems of the Web: the causes that make the Web unreliable.

1.1 The origin of the Web

The web was originally conceived and developed to meet the demand for automatic information-sharing between scientists in universities and institutes around the world[11].
The Web would then be quickly revealed as a communications tool to allow anyone, anywhere to share information.

1.1.1 The basic idea: A distributed HyperText

The basic idea was a hypertext whose documents are distributed on a computer network, that is "a distributed hypertext".

A client application, called *web browser*, obtains access to a hypertext document, called *web page*, stored on another computer, by sending a message over the network to a server application, called *web server*, on that computer, which in turn sends back the source code for the document.

An hypertext is a text which is not constrained to be linear, but contains links to other texts[12].
In a distributed hypertext texts are located in different computers connected each other.

Why the name HyperText

The term *hypertext* was coined by Ted Nelson in 1963[13].
The English prefix *hyper* comes from the Greek prefix "υπερ" and means "over" or "beyond"; it has a common origin with the prefix "super-" which comes from Latin; it signifies the overcoming of the linear constraints of written text.

The term *hypertext* is often used where the term *hypermedia*, which also was coined by Ted Nelson, might seem appropriate. However, the term "hypermedia", meaning complexes of branching and responding graphics, movies and sound – as well as text – is much less used.

[11] CERN. *The Birth of the Web* (https://goo.gl/4mYJqk)
[12] W3C. *What is HyperText* (https://goo.gl/M5VAbc)
[13] Ted Nelson. *Literary Machines* (https://goo.gl/tczxzL)

The origin of the HyperText

Tim Berners-Lee did not invent the hypertext, but he conceived of hypertext system distributed over a computer network.

In 1980, Tim Berners-Lee started a personal project to get to grips with managing information. He realized a software, that he called Enquire, named for a Victorian manual of domestic life called "*Enquire Within Upon Everything*"[14]:

> *It allowed one to store snippets of information, and to link related pieces together in any way. To find information, one progressed via the links from one sheet to another, rather like in the old computer game "adventure".*

By the 1960s, with the advancement of computer technology, this concept was implemented by pioneers such as Douglas Engelbart and Ted Nelson. They realized a program that allowed texts (or other media) to be viewed with some spans marked as hyperlinks, through which the reader could jump to another document[15].

Douglas Engelbart and Ted Nelson, for their part, were influenced by the ideas set out by Vannevar Bush. He proposed an organisation of external records (books, papers, photographs) corresponding to the association of ideas in human memory[16].

Vanner Bush, in turn, was no doubt influenced by the ideas of Belgian informatician Paul Otlet, who have been considered the father of information science, a field he called "documentation".

Each one of these giants in the history of hypertext was standing on the shoulders of the giants that had come before them.

---

[14] Bob Hopgood (2001). *History of the Web* (https://goo.gl/92khjf)
[15] Douglas Engelbart (1995). *Boosting our Collective IQ* (https://goo.gl/A3tD8k)
[16] Vannervar Bush (July 1945). *As we may think* (https://goo.gl/A6pemX)

1.1.2 The first Web

In 1990, aided and abetted by his colleague Robert Cailliau, Tim Berners-Lee built all the tools necessary for a working Web: the first web browser, which was a web editor as well, called "WorldWideWeb" (just one word)[17]; the first web server, which used a NeXT Computer; and so, the first web site, which described the Web project itself[18].

The first website at CERN - and in the world - was dedicated to the World Wide Web project itself. It described the basic features of the web, how to access other people's documents and how to set up your own server.

In 2013, as part of the project to restore the first website, CERN reinstated the world's first website to its original address[19].

The original web server — the Berners-Lee's NeXT computer — used to serve the first web site, is still at CERN.

[17] Tim Berners-Lee. *The WorldWideWeb browser* (https: //goo.gl/vzVTYk)
[18] CERN. *Home of the first website* (https://goo.gl/ifEU9R)
[19] CERN. *The Birth of the Web* (https://goo.gl/VjRRLS)

1.1.3 “This is for everyone”

In 1993, on April 30, CERN put the World Wide Web software in the public domain. CERN made the next release available with an open licence, as a more sure way to maximise its dissemination. Through these actions, making the software required to run a web server freely available, along with a basic browser and a library of code, the web was allowed to flourish[20].

The temptation to monetize this burgeoning hypertext system must have been hard to resist. But, perhaps inspired by the selfless spirit of cooperation and collaboration at CERN, Tim Berners-Lee and Robert Cailliau gave their gift to the world and asked for nothing in return[21].

In 2012, at the Summer Olympic games in London, Sir Tim Berners-Lee was lauded in the opening ceremony. Watched by a global audience, he passed on one message regarding the World Wide Web: “*This is for everyone.*”[22].

In 2017, Tim Berners-Lee received the Turing award for having invented the Web, the first web browser, and the fundamental protocols and algorithms that allowed the Web to scale[23].

Today we think of the World Wide Web as one of the greatest inventions in the history of communication, but to the scientists at CERN it is merely a byproduct. When you’re dealing in cosmological timescales and investigating the very building blocks of reality itself, the timeline of humanity’s relationship with technology is little more than a rounding error[24].

---

[20] K. Coldham (August 2016). "*Internaut Day and the World Wide Web*" (https://goo.gl/7vtsnb)
[21] M. Giampietro (April 2013). “*Twenty years of a free, open web*” (https://goo.gl/Sb2V3T)
[22] J. Keith (November 2013). “*This is for everyone*” (https://goo. gl/WofL58)
[23] J. Russell. (April 2017). “*Tim Berners-Lee, inventor of the world wide web, wins “computing’s Nobel Prize”*” (https://goo.gl/j6r9x5)
[24] J. Keith (2015). “*Resilient web design*” (https://goo.gl/ MvnUFv)

1.1.4 The Web and Internet

Informally people often use the terms Internet and Web interchangeably, but this is inaccurate. Even though they are closely related, the Web and Internet are two separate things. The Web is in fact just one of many services delivered over the Internet.

The Internet

The Internet is an extension of the technology of computer networks. It is a globally distributed networking infrastructure, a massive network of networks. It connects millions of computers together in the world, forming a network of networks, in which any computer can communicate with any other computer as long as they are both connected to the network. The internet has no centre. This architectural decision gives the network its robustness[25].

It's common to hear that the internet was designed to resist a nuclear attack, but that's not entirely correct. It's true that the project began with military considerations: the initial research was funded by DARPA, the Defense Advanced Research Projects Agency, but the engineers working on the project were not military personnel[26]. The underlying ideals had more in common with the free-speech movement than with the military-industrial complex. The Internet was designed to route around damage; even though the damage was not concerned a nuclear attack, but censorship.

The open architecture of the internet reflected the liberal worldview of its creators. As well as being decentralised, the internet was also deliberately designed to be a dumb network. The protocols underlying the transmission of data on the internet, the Transfer Control Protocol and the Internet Protocol (TCP/IP), describe how packets of data should be moved around, but those protocols care not a whit for the contents of the packets[27]. That decreted the success of Internet, allowing Internet to be the transport mechanism for all sorts of applications: email, file transfer, and eventually the World Wide Web.

---

[25] Vinton Cerf & Edward Cain (1983). "*The DoD Internet Architecture Model*" (https://goo.gl/pEzprW)
[26] Ronda Hauben. "*From the ARPANET to the Internet*" (https://goo.gl/JrMdDv)
[27] DARPA Internet Program (September 1981). RFC 793. "*Transmission Control Protocol. Protocol Specification*" (https://goo.gl/ZRS3Zb)

The Web

When Tim Berners-Lee was designing the Web at CERN, the Internet was already established as part of the infrastructure there. This network of networks was first created in the '60s and the early adopters were universities and scientific institutions.

The earliest computers operated independently. In the 1960s and 1970s, it became common for computers in an organisation (e.g., university, government, company) to be linked together in a network.

At the same time, there were early experiments in linking whole networks together, including the ARPANET in the United States.

In the early 1980s, the Internet Protocol Suite (TCP/IP) for the ARPANET was standardised, to provide the basis for a network of networks that could embrace the whole world[28].

The Internet spread mostly to Europe and Australia during the 1980s, and to the rest of the world during the 1990s.

The Web is an information-sharing model that is built on top of the Internet. It is just one of the ways that information can be disseminated over the Internet.

[28] DARPA Internet Program (September 1981). RFC 791. "*Internet Protocol. Protocol Specification*" (https://goo.gl/G7y15m)

1.2 The protocols of the Web

Any communication between two entities requires the two entities agree on the way and the language used to communicate. All communications between two nodes on the Internet require the two nodes agree on the way and the format of the data they exchange to each other. The set of rules defining a format is called a protocol.

The Web has a body of software, and a set of protocols and conventions. The protocols of the Web rely on the protocols of the Internet network it is built on.

1.2.1 The Internet Protocol suite

The technology supporting the Internet includes the IP (Internet Protocol) system for addressing computers, so that messages can be routed from one computer to another.
Each computer on the Internet is assigned an IP address.

The structure of messages is governed by application protocols that vary according to the service required[29]. For example the SMTP (Simple Mail Transfer Protocol) for email[30], the FTP (File Transfer Protocol) for file transfer[31], and HTTP (HyperText Transfer Protocol) for the Web[32].

The Internet Protocol (IP) is complemented by the Transmission Control Protocol (TCP), originated in the initial network implementation. Therefore, the entire protocol suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of data bytes between applications running on hosts communicating by an IP network.

Applications that do not require reliable data stream service may use the User Datagram Protocol (UDP), which provides a connectionless datagram service that emphasizes reduced latency over reliability[33].

---

[29] IETF (October 1989). RFC 1123. "*Requirements for Internet Hosts - Application and Support*" (https://goo.gl/s3yjnf)
[30] Jonathan B. Postel (August 1982). RFC 821. Simple Mail Transfer Protocol (https://goo.gl/C2XTLT)
[31] IETF (October 1985). RFC 959. "*File Transfer Protocol (FTP)*" (https://goo.gl/V4QZKi)
[32] UC Irvine & J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. (June 1999). RFC 2616. "*Hypertext Transfer Protocol - HTTP/1.1*" (https://goo.gl/PGcDBi)
[33] J. Postel (August 1980). RFC 768: User Datagram Protocol (https://goo.gl/rgTRPN)

1.2.2 The HyperText Transfer Protocol (HTTP)

The Web uses the HyperText Transfer Protocol (HTTP), to send and receive data. HTTP defines how messages are formatted and transmitted between web servers and web browsers, and what actions web servers and web browsers should take in response to various commands.

Resources in the Web are uniquely identified by global identifiers called Uniform Resource Identifiers (URI)[34].

Many of these URIs identify documents written in the HyperText Markup Language (HTML), called web pages[35].

Webpages can contain hyperlinks to other resources on the Web, as well as other webpages, enabling the distributed hypertext.

When the user clicks on a hyperlink, the web browser finds the IP address associated with the URL[36], and sends a HTTP message to this IP address requesting the HTML file at the given location in the server's file system; on receipt, this file is displayed in the web browser.

Links in web pages turned the Web from being a straightforward storage and retrieval system into a world wide distributed hypertext system. Through the use of hypertexts and multimedia techniques, and leveraging on HTTP, the Web has been allowing anyone to roam, browse, and contribute to.

---

[34] T. Berners-Lee, R. Fielding, Day Software & L. Masinter. (January 2005). RFC 3986. "*Uniform Resource Identifier (URI): Generic Syntax*." (https://goo.gl/NnQh6N)
[35] Tim Berners-Lee & Daniel Connolly. (June 1993). Internet Draft. "*Hypertext Markup Language (HTML) - A Representation of Textual Information and Meta Information for Retrieval and Interchange*." (https://goo.gl/y8GZsm)
[36] T. Berners-Lee, L. Masinter, M. McCahill. (December 1994). RFC 1738. Uniform Resource Locators (URL). (https://goo.gl/TuuhoY)

1.3 The languages of the Web

1.3.1 The Hypertext Markup Language (HTML)

The Hypertext Markup Language (HTML) is the standard markup language for creating web pages[37]. HTML describes the structure of a web page semantically, though originally included cues for the appearance of the document[38].

HTML isn't the first markup language to be used. When Tim Berners-Lee designed the Web at CERN, scientists there were already sharing documents written in SGML (Standard Generalized Markup Language)[39]. Tim Berners-Lee used the SGML as a starting point for a new markup language, the HTML (HyperText Markup Language). It made sense to build something on what people were already familiar with rather than creating something from scratch [12].

HTML provides a means to create structured documents. An HTML document is composed of a hierarchical set of nodes. Each node can have HTML attributes specified, include text as well as other nodes.

Syntax

In the HTML syntax, most nodes are written with a start tag and an end tag, with the content in between. An HTML tag is composed of the name of the node, surrounded by angle brackets `<` and `>`. An end tag also has a slash `/` after the opening angle bracket, to distinguish it from the start tag[40].

For example:

```
<node-name>content</node-name>
```

Where:

- `<node-name>` is the start tag
- `</node-name>` is the end tag

[37] WhatWG (June 2018). "*HTML Living standard*". (https://goo.gl/cuYJBX)
[38] W3C (December 2017). W3C Recommendation: "*HTML 5.2*" (https://goo.gl/g1F8GP)
[39] Dan Connolly (January 1996). "*A Study of Linguistics: Representation and Exchange of Knowledge*" (https://goo.gl/JU4yDG)
[40] WhatWG. (June 2018). "*The HTML syntax*". (https://goo.gl/59rdAN)

Structure

An HTML document has a well defined structure. The element `html` is the root that contains just two sub-elements:

- `head` is the container for processing information, such as the title of the document, and metadata, such as the description of the document;
- `body` is the container for the displayable content of the HTML document[41].

For example:

```
<html>
  <head>
    <meta description="This is the description">
    <title>This is the title</title>
  </head>
  <body>
    <h1>This is a title</h1>
    <h2>This is a subtitle</h2>
    <p>This is an <em>emphasized</em> paragraph</p>
  </body>
</html>
```

Representation

An HTML document is delivered as textual document[42]. The web browser parses the HTML document and turns it into an internal representation, called Document Object Model (also known as the DOM)[43].

Presentation by the web browser is performed on this internal object model, not the original textual document. The nodes of the textual document are called *tags*, while the corresponding representations of the DOM are called *elements*. HTML documents contain tags, but do not contain the elements. The elements are only generated after the parsing step, from these tags.

[41] W3C. '*The global structure of an HTML document*" (https://goo.gl/pdSGVF)
[42] W3C. "*HTML Document Representation*" (https://goo.gl/1sYPpG)
[43] W3C. "*Document Object Model (DOM)*" (https://goo.gl/WudRXo)

Semantic

Some HTML elements are literally meaningless, like the element `span` that says nothing about the contents within it[44]. But most HTML elements exist for a reason: they have been created and agreed upon in order to account for specific situations[45].

There are special elements, like the element `a`, that allows users to link out to any other resource on the web, or like the elements `input`, `select`, `textarea`, and `button`, that allows users to enter data and submit it to a web server.

There are elements that describe the kind of content they contain.
For example:

- element `p` is a paragraph.
- element `ol` is an ordered (numbered) list of items.
- element `ul` is an unordered (bullet) list of items.
- element `li` is an item in a list (be it ordered or not).

Browsers display these elements with some visual hints as to their meaning.
For example, anchors are underlined; paragraphs are displayed with whitespace before and after their content; ordered list items are prefixed with numbers, while unordered list items are prefixed with bullet points.

The early growth of HTML’s vocabulary was filled with new elements that provided visual instructions to web browsers: `big`, `small`, `center`, `font`, etc. In fact, the visual instructions were the only reason for those elements to exist, they provided no hint as to the meaning of the content they contained. HTML was in danger of becoming a visual instruction language instead of a vocabulary of meaning.

[44] W3C. The first introduction to “*HTML tags*”. (https://goo.gl/E9NALU)
[45] MDN. "*HTML elements reference*". (https://goo.gl/L7WTQi)

1.3.2 The Cascading Style Sheet (CSS)

Håkon Wium Lie was working at CERN at the same time as Tim Berners-Lee. He immediately recognised the potential of the World Wide Web and the HTML. He also realised that the expressive power of the language was in danger of being swamped by visual features. He proposed a new format to describe the presentation of HTML documents: the Cascading Style Sheets (CSS)[46].

Bert Bos quickly joined Lie. They set about creating a syntax that would be powerful enough to handle the demands of designers, while remaining simple enough to learn quickly.

There's a huge variation in visual style on the Web, such as colour schemes, typographic treatments, layouts, etc. All of the styling variety is made possible by one simple pattern that describes all the CSS ever written.

```
selector {
  property: value;
}
```

Actually, CSS specifications is composed by different levels[47].

Separation of content and presentation

Separation of content and presentation or separation of content and style, is a design principle under which visual and design aspects (presentation and style) are separated from the core material and structure (content) of a document.

This principle is not a rigid guideline, but serves more as best practice for keeping appearance and structure separate. In many cases, the design and development aspects of a project are performed by different people, so keeping both aspects separated ensures both initial production accountability and later maintenance simplification, as in the don't repeat yourself (DRY) principle[48].

[46] H. W. Lie. (1994). "*Cascading Style Sheets*". (https://goo.gl/t5xgKm)
[47] W3C. "*Description of all CSS specifications*". (https://goo.gl/sQLd1t)
[48] W3C. (June 2003). "*Separation of semantic and presentational markup, to the extent possible, is architecturally sound*". (https://goo.gl/FKroGV)

The loose error-handling

HTML and CSS have a forgiving attitude to errors.
Echoing the Robustness Principle[49], also known as Postel's Law:

> *Be conservative in what you send; be liberal in what you accept.*

Even if there are errors in the HTML or CSS, the browser will still attempt to process the information, skipping over any pieces that it can't parse. The browser doesn't throw an error. The browser doesn't stop parsing, refusing to go any further.

If the browser sees an HTML element it doesn't know, it just ignores the element but continues to parse its content. If the browser sees an HTML attribute for an element it doesn't know, it just ignore the attribute.

```
<unknown>This tag is ignored</unknown>
<a href="/" not-yet-defined-attribute="value">link<a>
```

If the browser sees a CSS selector it doesn't understand, it just ignores that selector's styling rules. If the browser sees a CSS property or a value it doesn't understand, it just ignores that particular declaration.

```
unknown {
  not-yet-defined-property: value;
}
```

The loose error-handling has allowed CSS to grow over time. New selectors, new properties, and new values have been added to the language's vocabulary over the years. Whenever a new feature lands in CSS, designers and developers know that they can safely use it, even if it isn't yet widely supported in browsers. They can rest assured that old browsers will react to new features with complete indifference.

[49] Wikipedia. *Robustness principle*. (https://goo.gl/7dm4K7)

Declarative languages

HTML and CSS are both examples of declarative languages.

A declarative language describes a desired outcome without providing step-by-step instructions to the program that parse the document[50].

An HTML document describes the nature of the content, such as headings, paragraphs, lists, etc., without having to explain to the browser, that parses the document, exactly what it should do display those content.

A CSS document describes the desired appearance of the content, such as colors, positions, dimensions, etc., without having to explain to the browser, that parse the document, exactly what it should do to apply those styles.

Most programming languages are not declarative, but imperative.

An imperative language provides precise instructions to the program that interprets the code. Imperative languages provide more power and precision than declarative languages, but at a price: imperative languages tend to be harder to learn than declarative languages.

Furthermore, they make it harder to apply Postel's law. If you make a single mistake - even a misplaced comma - the entire program may fail. A misspelt tag in HTML or a missing curly brace in CSS can also cause headaches, but imperative programs must be well-formed or they won't run at all.

Imperative languages such as PHP, Ruby, and Python can be found on the servers powering the World Wide Web, reading and writing database records, processing input, and running complex algorithms. You can choose just about any programming language you want when writing server-side code.

If you want to write code that runs in a web browser, you only have one choice: JavaScript.

[50] Wikipedia. Declarative programming (https://goo.gl/Y7EbwR)

1.3.3 JavaScript

The idea of executing a program from within a webpage is as old as the Web itself. From an email to the www-talk mailing list dated May 1992[51]:

> *I would like to know, whether anybody has extended WWW such, that it is possible to start arbitrary programs by hitting a button in a WWW browser.*

Tim Berners-Lee responded:

> *Very good question. The problem is that of programming language. You need something really powerful, but at the same time ubiquitous. Remember a facet of the web is universal readership. There is no universal interpreted programming language. But there are some close tries. (lisp, sh). You also need something which can run in a very safe mode, to prevent virus attacks.*
>
> *Ideally, the language should include object-oriented inheritance, a basically functional nature, and a clean syntax. It should be interpretable and compilable. At least one public domain. A pre-compiled standard binary form would be cool too.*
>
> *It isn't here yet.*

A universal interpreted programming language wasn't there yet, until 1996.
In 1996, Brendan Eich, a programmer at Netscape, wrote it, in ten days[52].

Why the name JavaScript

The language went through a few name changes. First, it was called *Mocha*. Then, it was launched as *LiveScript*. Finally, it was renamed as *JavaScript*, to ride the wave of hype associated with the then-new *Java* language. Even though JavaScript and Java have little in common.

---

[51] Tim Berners-Lee (May 1992). "*Program Links in WWW*" (https://goo.gl/JDBPZB)
[52] Brendan Eich (June 2005). "*JavaScript 1, 2, and in between*". (https://goo.gl/AhNVLs)

The use of JavaScript

JavaScript gave web designers the power to create websites that were slicker, smoother, and more reactive. The same technology also gave web designers the power to create websites that were sluggish, unwieldy, and more difficult to use.

Being it an imperative language, if you give a web browser some badly-formed JavaScript or attempt to use an unsupported JavaScript feature, not only will the browser throw an error, it will stop parsing the script at that point and refuse to go any further.

The DOM

The Document Object Model (DOM) is a programming interface for HTML documents. It represents the page so that programs can change the document structure, style, and content[53].

A Web page is a document. The DOM is an object-oriented representation of the web page: it represents the document as nodes and objects that can be modified with a scripting language.

The W3C DOM and WHATWG DOM[54] standards are implemented in most modern browsers in JavaScript.

For example:

```
let heading = document.createElement('h1')
let heading_text = document.createTextNode('Hello DOM!')
heading.appendChild(heading_text)
document.body.appendChild(heading)
```

[53] MDN. "*Introduction to the DOM*" (https://goo.gl/hFL7NB)
[54] WhatWG. "*DOM Living Standard*". (https://goo.gl/rdaCnm)

The Asynchronous JavaScript and XML (AJAX)

At the beginning, JavaScript was just used for creating visual effect, such as rollover, and form validation. Swapping out images when someone hovers their cursor over a link might not seem like a sensible use of a brand new programming language, but back in the nineties there was no other way of creating hover effects.

Both of those use cases still exist, but now there's no need to reach for JavaScript. You can create rollover effects in CSS, and you can validate form fields using the new functionalities of the HTML input element.

This reveals a common pattern: a solution is created in an imperative language and if it's popular enough, it migrates to a declarative language over time. When a feature is available in a declarative language, not only is it easier to write, it's also more robust.

The rise of JavaScript was boosted in 2005 with the publication of an article entitled Ajax: A New Approach to Web Applications by Jesse James Garrett[55].
The article put a name to a technique that was gaining popularity.

With this technique, using a specific subset of JavaScript, it was possible for a web browser to send and receive data from a web server asynchronously (in the background) without interfering with the display and behaviour of the existing web page. By decoupling the data interchange layer from the presentation layer, it allows for web pages to change content dynamically without the need to reload the entire page.

The term Ajax is a short for "*asynchronous JavaScript and XML*", it has come to represent a broad group of Web technologies[56]:

- HTML and CSS for presentation
- The DOM for dynamic display of and interaction with data
- XML for the interchange of data
- The XMLHttpRequest JavaScript object for asynchronous communication
- JavaScript to bring these technologies together

[55] J. J. Garrett (February 2005). "*Ajax: A New Approach to Web Applications*". (https://goo.gl/t4r6oz).
[56] MDN. "*Ajax*". (https://goo.gl/ktuCo7)

1.3.4 The Extensible Markup Language (XML)

The Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
The W3C's XML 1.0 Specification[57] and several other related specifications —all of them free open standards— define XML. XML is claimed as a self-describing language, though the XML specification itself makes no such claim.

Hundreds of document formats using XML syntax have been developed, including RSS, Atom, SOAP, SVG, and XHTML. XML-based formats have become the default for many office-productivity tools, including Microsoft Office (Office Open XML), OpenOffice.org and LibreOffice (OpenDocument). XML has also provided the base language for communication protocols such as XMPP.

XML and its extensions have regularly been criticized for verbosity and complexity. Mapping the basic tree model of XML to type systems of programming languages or databases can be difficult, especially when XML is used for exchanging highly structured data between applications, which was not its primary design goal.

JSON is frequently proposed as simpler alternative, that focus on representing highly structured data rather than documents, which may contain both highly structured and relatively unstructured content.

[57] W3C. (November 2008). "*Extensible Markup Language (XML) 1.0 (Fifth Edition)*". (https://goo.gl/VydyAv)

1.3.5 The JavaScript Object Notation (JSON)

JavaScript programming language carried on with it a data format, JSON (JavaScript Object Notation)[58].

Data Types

JSON has four basic data types[59]:

- number
- string
- boolean
- null

and two complex data types:

- array
- object

Number

The format makes no distinction between integer and floating-point. A signed decimal number that may contain a fractional part and may use exponential E notation, but cannot include non-numbers. JavaScript uses a double-precision floating-point format for all its numeric values, but other languages implementing JSON may encode numbers differently.

```
123
3.141592653589793
1E9
```

String

A string is a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.

```
"string"
"a string \n in two rows"
```

[58] ECMA (December 2017). Standard ECMA-404. "*The JSON Data Interchange Syntax*" (https://goo.gl/qQ9ZZm)
[59] json.org. "*Introducing JSON*" (https://goo.gl/giEeWz)

Boolean

A boolean is either of the values true or false.

```
true
false
```

Null

A null is empty value, using the word null.

```
null
```

Array

An array is an ordered list of zero or more values, each of which may be of any type. Arrays are delimited with square bracket and values are comma-separated.

```
[1, 1.4142135623730951, 2, 3, 3.141592653589793, 4]
["a", 2, true, null]
["a", ["another array", 42]]
```

Object

An object is an unordered collection of name–value pairs where the names (also called keys) are strings. Since objects are intended to represent associative arrays, it is recommended, though not required, that each key is unique within an object.
Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon character separates the key or name from its value.

```
{
  "key": "value",
  "another key": null,
  "nested object": {
    "array": [1, 2, 3]
  }
}
```

1.4 The evolution of the Web

During the years, the Web has evolved to influence how we interact, how we keep informed, how we keep learning, in general, how we proceed day to day.
For many years, the Web was a "read-only" tool for many.

1.4.1 The static Web (the Web 1.0)

In 1993 came a turning point for the WWW with the introduction of the Mosaic web browser, which could display graphics as well as text[60].

From that date, usage of the web grew rapidly, although most users operated only as consumers of content, not producers.

Enter Web 1.0. During this early phase of web development, web pages were mostly static documents read from a server and displayed on a client, with no options for users to contribute content, or for content to be tailored to a user's specific demands.

[60] NCSA. "*NCSA Mosaic™*" (http://www.ncsa.illinois.edu/enabling/mosaic)

1.4.2 The dynamic Web (the Web 2.0)

Around 2000 a second phase of web development began with the increasing use of technologies allowing the user of a browser to interact with web pages and shape their content. Tim O'Reilly used the phrase Web 2.0 to describe a new wave of web products and services[61].

It was difficult to pin down a definition of Web 2.0. For business people, it meant new business models. For graphic designers, it meant rounded corners and gradients. For developers, it meant JavaScript and Ajax.

Whatever its exact meaning, the term Web 2.0 captured a mood and a feeling. Everything was going to be different now. The old ways of thinking about building for the web could be cast aside. Treating the web as a limitless collection of hyperlinked documents was passé. The age of web apps was at hand.

Web 2.0 represented the move toward a more social, collaborative, interactive and responsive web. It served as a marker of change in the philosophy of web companies and web developers. Even more than that, Web 2.0 was a change in the philosophy of a web savvy society as a whole. Both the change in how society functions as well as the internet as an existing form of technology are part of Web 2.0. Web 2.0 marked an era where we weren't just using the internet as a tool anymore — we were becoming a part of it. Web 2.0 is the process of putting "us" into the web.

[61] Tim O'Reilly (September 2005). "*What Is Web 2.0 - Design Patterns and Business Models for the Next Generation of Software*". (https://goo.gl/JwgeCE)

1.4.3 The social Web

Web 2.0 technologies have made possible a wide range of social web sites now familiar to everyone, including chat rooms, blogs, wikis, product reviews, e-markets, and crowdsourcing.

Previously a consumer of content provided by others, the web user has now become a prosumer, capable of adding information to a web page, and in this way communicating not only with the server, but through the server with other clients as well.

The idea of human society that blends with a computer network looks like a science fiction scene anticipated by a dreamy past, but it is a fair description of what has happened to our society at the beginning of the new millennium.

Not only have we increased our usage of the internet, from how much time we started spending on it at home to how we now carry around a version of it in our pocket, but we have changed the way we interact with it.

This has led us to a social web where we aren't just getting information dumped onto us from a computer because we're now all connected with other people who can put anything they want to share online.

Nowadays, the Web is the main communication channel of everyday life.
We are getting more and more connected thanks to the Web. In just few decades we've tremendously extended our boundaries and shortened our distances: we can speak to half the planet, at any time and from almost anywhere; we can examine the most complete compendium of human knowledge in seconds; we can work, study, play, and be together at a distance.

1.5 The Semantic Web

Tim Berners-Lee original idea of the Web concealed another idea, a Web of data.
That idea would become the Semantic Web.

The Semantic Web extends the network of hyperlinked human-readable web pages by inserting machine-readable metadata about pages and how they are related to each other. This enables automated agents to access the Web more intelligently and perform more tasks on behalf of users.

The Semantic Web is about two things. It is about common formats for integration and combination of data drawn from diverse sources, where on the original Web mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing[62].

The 2001 Scientific American article by Tim Berners-Lee, Hendler, and Lassila described an expected evolution of the existing Web to a Semantic Web[63]. In 2006, Berners-Lee and colleagues stated that: "*This simple idea…remains largely unrealized*". However, in 2013, more than four million Web domains contained Semantic Web markup.[64]

---

[62] W3C. (November 2011). "*W3C Semantic Web activity*". (https://goo.gl/2otfYn)
[63] Tim Berners-Lee, James Hendler and Ora Lassila. (May 2001). "*The Semantic Web - a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities.*" (https://goo.gl/Kxg3ud)
[64] Ramanathan V. Guha (March 2015). "*Light at the End of the Tunnel*".
(https://goo.gl/YzijhY)

1.5.1 Microdata

Microdata[65] is a set of tags (HTML attributes), introduced with HTML5, to help search engines and other applications better understand web content and display it in a useful, relevant way.

Microdata provides a simple mechanism to label content in a document, so it can be processed as a set of items described by name-value pairs. Each name-value pair identifies a property of the item, and a value of that property.

Itemscope and itemtype

Items and properties are generally represented by regular elements.
The `itemscope` attribute on an element identifies the element as an item.
The `itemprop` attribute on a descendant of an item identifies a property of that item.

Typically, the text content of that element is the value of that property.
If text content is not suitable to contain the value the `content` attribute of the element is used.

For example:

```
<div itemscope itemtype="http://schema.org/Movie">
  <h1 itemprop="name">Avatar</h1>
  <div itemprop="director" itemscope itemtype="http://schema.org/Person">
    <span>Directed by<span> <span itemprop="name">James Cameron</span>
  </div>
  <span itemprop="genre">Science fiction</span>
  <a href="/movies/avatar-trailer" itemprop="trailer">Trailer</a>
</div>
```

Many pages can be described using only the `itemscope`, `itemtype`, and `itemprop` attributes along with the types and properties defined on schema.org[66].
Most sites and organizations will not have a reason to extend schema.org. However, schema.org offers the ability to specify additional properties or sub-types to existing types[67].

[65] W3C (April 2018). *HTML Microdata*. (https://goo.gl/74i7NJ)
[66] Schema.org. *Getting started with schema.org using Microdata* (https://goo.gl/c1MJBm)
[67] Schema.org. *Schema.org Extensions*. (https://goo.gl/GuuwrV)

1.5.2 Schema.org

Schema.org is a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet, on web pages, and beyond[68].

Web pages have an underlying meaning that people understand when they read them. But search engines have a limited understanding of what is being discussed on those pages.

Schema.org provides a collection of shared vocabularies to mark up web pages in ways that can be understood by the major search engines: Google, Microsoft, Yandex and Yahoo!

Schema.org vocabulary can be used with many different encodings, including RDFa, Microdata and JSON-LD[69]. These vocabularies cover entities, relationships between entities and actions, but can be extended.

Schema.org is used by over 10 million sites to markup their web pages. Many applications from Google, Microsoft, Pinterest, Yandex and others already use these vocabularies to power rich, extensible experiences.

Founded by Google, Microsoft, Yahoo and Yandex, Schema.org vocabularies are developed by an open community process[70], using the public-schemaorg@w3.org mailing list and through GitHub platform.

A shared vocabulary makes it easier for webmasters and developers to decide on a schema and get the maximum benefit for their efforts. It is in this spirit that the founders, together with the larger community have come together - to provide a shared collection of schemas.

---

[68] Schema.org. *Welcome to Schema.org*. (https://goo.gl/AVBrs5)
[69] W3C. (January 2014). *JSON-LD 1.0 - A JSON-based Serialization for Linked Data*. (https://goo.gl/yNWkwN)
[70] W3C. *Schema.org community group* (https://goo.gl/6PBsqb)

Types and properties

Schema.org describes a variety of other item types, each of which has its own set of properties that can be used to describe the item.

The broadest item type is `Thing`[71], which has four properties: name, description, url, and image.

More specific types share properties with broader types.
For example, a `Place` is a more specific type of `Thing`, and a `LocalBusiness` is a more specific type of `Place`.

More specific items inherit the properties of their parent.
For example, a `LocalBusiness` is a more specific type of `Place` and a more specific type of Organization, so it inherits properties from both parent types.

The commonly used item types are: creative works such as `Book`, `Movie`, `Music`, `Recipe`, etc.; events and people such as `Event`, `Organization`, `Person`, `Place` and embedded non-text objects, such as `AudioObject`, `ImageObject`, `VideoObject`, etc.

---

[71] Schema.org. *Schema.org extensions*. (https://goo.gl/TTkHXY)

1.5.3 Open Graph Protocol

The Open Graph protocol enables any web page to become a rich object in a social graph[72].
It is used on Facebook to allow any web page to have the same functionality as any other object on Facebook.

While many different technologies and schemas exist and could be combined together, there isn't a single technology which provides enough information to richly represent any web page within the social graph. The Open Graph protocol builds on these existing technologies and gives developers one thing to implement.[73]

Basic Metadata

To turn web pages into open graph objects, you need to add basic metadata.
The required metadata properties for every page are:

- `og:title` - the title of your object as it should appear within the graph;
- `og:type` - the type of your object;
- `og:image` - an image URL which should represent your object within the graph;
- `og:url` - the canonical URL of your object that will be used as its ID in the graph.

For example:

```
<html prefix="og: http://ogp.me/ns#">
  <head>
    <title>The Rock (1996)</title>
    <meta property="og:title" content="The Rock" />
    <meta property="og:type" content="video.movie" />
    <meta property="og:url"
          content="http://www.imdb.com/title/tt0117500/" />
    <meta property="og:image"
          content="http://ia.media-imdb.com/images/rock.jpg" />
    ...
  </head>
...
</html>
```

[72] Open Web Foundation. *The Open Graph protocol* (https://goo.gl/CkzZJC)
[73] Facebook. *The Open Graph Protocol Design Decisions*. (https://goo.gl/R8hi2K)

Optional Metadata

The following properties are optional for any object and are generally recommended:

- `og:audio` - a URL to an audio file to accompany this object.
- `og:description` - a one to two sentence description of your object.
- `og:determiner` - the word that appears before this object's title in a sentence.
- `og:locale` - the locale these tags are marked up in.
- `og:locale:alternate` - an array of other locales this page is available in.
- `og:site_name` - the name of the web site the object belongs to.
- `og:video` - a URL to a video file that complements this object.

Structured metadata

Some properties can have extra metadata attached to them. These are specified in the same way as other metadata with property and content, but the property will have extra :.

For example, the og:image property has some optional structured properties:

- `og:image:url` - identical to og:image.
- `og:image:secure_url` - an alternate url to use if the webpage requires HTTPS.
- `og:image:type` - a MIME type for this image.
- `og:image:width` - the number of pixels wide.
- `og:image:height` - the number of pixels high.
- `og:image:alt` - a description of what is in the image (not a caption).

Types

The types used when defining attributes are:

- Boolean: a true or false value: true, false, 1, 0.
- DateTime: a temporal value composed of a date and a time, in ISO 8601[74].
- Enum: a set of constant string values (enumeration members).
- Float: a 64-bit signed floating point number.
- Integer: a 32-bit signed integer.
- String: a sequence of Unicode characters (with no escape characters)
- URL: a sequence of Unicode characters that identify an Internet resource.

[74] Wikipedia. *ISO 8601* (https://goo.gl/6s2N7K)

1.5.4 AMP

AMP HTML is a subset of HTML for authoring content pages such as news articles in a way that guarantees certain baseline performance characteristics[75].
Being a subset of HTML, it puts some restrictions on the full set of tags and functionality available through HTML but it does not require the development of new rendering engines: existing user agents can render AMP HTML just like all other HTML.

AMP HTML documents can be uploaded to a web server and served just like any other HTML document: no special configuration for the server is necessary. However, they are designed to be optionally served through specialized AMP serving systems that proxy AMP documents, applying transformations that provide additional performance benefits, such as:

- replace image references with images sized to the viewer's viewport;
- inline images that are visible above the fold;
- inline CSS variables;
- preload extended components;
- minify HTML and CSS.

AMP HTML uses a set of contributed but centrally managed and hosted custom elements to implement advanced functionality such as image galleries. While it does allow styling the document using custom CSS, it does not allow author written JavaScript beyond what is provided through the custom elements to reach its performance goals.

By using the AMP format, content producers are making the content in AMP files available to be crawled (subject to robots.txt restrictions), cached, and displayed by third parties.

AMP HTML documents should be annotated with standardized metadata, such as the Open Graph Protocol, and marked up with schema.org CreativeWork or any of its more specific types such as NewsArticle or BlogPosting.

---

[75] Google. *AMP HTML Specification*. (https://goo.gl/1SWGYd)

The AMP HTML format

HTML tags can be used unchanged in AMP HTML.
Certain tags have equivalent custom tags, such as `<amp-img>` for `<img>`, `<amp-video>` for `<video>`, `<amp-audio>` for `<audio>`, while other tags are outright prohibited, such as `<script>` (unless the type is application/ld+json), `<base>`, `<frame>`, `<frameset>`, `<object>`, `<param>`, `<applet>`, and `<embed>`.

For example:

```
<!doctype html>
<html amp>
  <head>
    <meta charset="utf-8">
    <title>Sample document</title>
    <link rel="canonical" href="./regular-html-version.html">
    <meta name="viewport"
      content="width=device-width, minimum-scale=1, initial-scale=1">
    <style amp-custom>/* AMP style */</style>
    <script type="application/ld+json">
    {
      "@context": "http://schema.org",
      "@type": "NewsArticle",
      "headline": "Article headline",
      "image": ["thumbnail-1.jpg", "thumbnail-2.jpg"],
      "datePublished": "2018-09-16T07:00:00+01:00"
    }
    </script>
    <script async custom-element="amp-carousel"
      src="https://cdn.ampproject.org/v0/amp-carousel-0.1.js"></script>
    <style amp-boilerplate>/* AMP style */</style>
    <noscript><style amp-boilerplate>/* AMP style */</style></noscript>
    <script async src="https://cdn.ampproject.org/v0.js"></script>
  </head>
  <body>
    <h1>Sample document</h1>
    <p>Some text</p>
    <amp-img src=sample.jpg width=300 height=300></amp-img>
    <amp-ad width=300 height=250 type="a9" data-aax_size="300x250"
      data-aax_pubname="test123" data-aax_src="302"></amp-ad>
  </body>
</html>
```

1.6 The problems of the Web

1.6.1 The Inefficient Web

The Web is inefficient. Even if a thousand people have downloaded a thousand copies of the same file to a thousand different physical locations, all references to that file would still point to that original, single location.

As of this writing, the most viewed video of all time on Youtube has over 5 billions views[76].

Let's make some assumptions. The video clocks in at 100 Mb, that means: at most 500 Petabytes of data for the video file alone has been sent since this was published.
If we assume a total expense of 1 cent per gigabyte, that would include bandwidth and all of the server costs, $5 Millions has been spent on distributing this one file so far.

HTTP lowered the price of publishing, but it still costs money, and these costs can really add up. Distributing this much data from central datacenters is potentially very expensive if not done at economies of scale.

[76] Youtube. *The most viewed video of all time*. (https://goo.gl/VhtkGM)

1.6.2 The Lost Web

The so idealized “Web of documents” actually is a “Web of documents on specific machines”.

If a document is moved in another location or just renamed, any link to that document is no longer traversable. Also, if the server is turned off, or if it’s made inaccessible, in any way, for whatever reason, any document hosted by that server is no longer available.

Paraphrasing Thomas Reid “*a chain is as strong as the weakest ring*”[77], location-addressing approach result to be the weak ring of the chain. Unreliable, breakable links lead to an unreliable, broken Web.

If “*a picture picture is worth a thousand of words*”[78], the following detailed flowchart shows the problem in detail.

404

Also, if a document is updated, remaining in the same location, the previous version of the document is no longer accessible: the Web only exists in the land of the perpetual present.

As far back as 2003, a large-scale study of the evolution of web pages[79], discovered that about one link out of every 200 disappeared each week from the Web, leaving the average lifespan of a web page at just 100 days[80].

[77] *The meaning and origin of the expression: A chain is only as strong as its weakest link.* (https://goo.gl/Ya8Wws)
[78] University of Regina. *The history of a picture's worth.* (https://goo.gl/q3Tg6f)
[79] Dennis Fetterly, Mark Manasse, Marc Najork, Janet Wiener. *A Large-Scale Study of the Evolution of Web Pages.* (https://goo.gl/ZStQgx)
[80] Mike Ashenfelder. (November 2011). The Average Lifespan of a Webpage. (https://goo.gl/nWqQGa)

1.6.3 The censored Web

Over the past decade, we have witnessed many attempts to erode the civil liberties, to censor the freedom of information, in the real world, and so even on the Web.

At 8am local time on April 29th, 2017, Wikipedia went dark for everyone in Turkey. According to the independent watchdog group Turkey Blocks, the Turkish government has issued a court order that permanently restricts access to the online encyclopedia[81].

The Chinese government has blocked Wikipedia[82], the New York Times[83], and other sites from its citizens[84]. And so do other countries every once in a while[85].

In January 2016, all content on Medium has been unavailable for Malaysian internet users[86].

In April 2016, Medium was blocked in mainland China after information from the leaked Panama Papers was published on the site[87].

In June 2017, Medium has been blocked in Egypt along with more than 60 online media websites in a crackdown by the Egyptian government. The list of blocked sites also includes Al Jazeera, the Huffington Post's Arabic website and Mada Masr[88].

The Web is for everyone, but not everyone is for the Web.

---

[81] D. Morris. (April 2017). *Turkey Blocks All Versions of Wikipedia*. (https://goo.gl/tzkw2s)
[82] J. Li and C. Wickenkamp. (June 2015). *Chinese Regime China Now Blocked From Accessing Wikipedia*. (https://goo.gl/d2unFs)
[83] K. Bradsherd. (December 2008). *China Blocks Access to The Times's Web Site*. (https://goo.gl/ntpy2r)
[84] B. Xu and E. Albert. (February 2017). *Media Censorship in China*. (https://goo.gl/pQ3U83)
[85] E. Schmidt and J. Cohen. (April 2013). *Web censorship: the net is closing in*. (https://goo.gl/hAjTUj)
[86] Medium. *The Post Stays Up*. (https://goo.gl/hqYdhA)
[87] Steven Millward. (April 2016). *Medium is now blocked in China*. (https://goo.gl/CCWS1H)
[88] Al Jazeera. *Egypt bans Medium as media crackdown widens*. (https://goo.gl/UqiJ6w)

1.6.4 The Offline Web

Usually, if you have no Internet connection, you can not navigate the Web, built over Internet. Techniques such as caching, through Service Workers, in HTML5 compliant web browsers, tries to alleviate the problems, but, in general, the Web is not built to live offline.

The following picture shows the problem in details.

That is: the on-line-or-no-live Web.

1.6.5 The Centralized Web

When the Web first took off in the mid '90s, the dream wasn't just big, it was distributed. The Web consisted of nodes joined by links, with no center. Everyone would have their own homepage, everyone would post their writings, everyone would own their own data, and there was no one around offering to own it for them. Without asking anyone permission, everyone could "*put information onto the Web*"[89].

But as the Web has grown into a global medium for communication, journalism, and entertainment, the power dynamics have shifted. Now, a handful of companies own vast swaths of web activity, Google for searching, Facebook for social networking, Amazon for e-commerce, and quite literally own the data their users have provided and generated.

This gives these companies unprecedented power over users, and gives them such a competitive advantage that it's not realistic to think to start up a business that's going to beat them at their own game.

[89] W3C. (April 1995). *Putting information into the Web*. (https://goo.gl/uaKd4G)

7.3 The attempts to re-decentralize the Web

There have been many attempts to re-decentralize the Web.

Tim Berners-Lee is leading a new project at MIT, that aims to radically change the way Web apps work today, resulting in true data ownership as well as improved privacy[90].

Solid (derived from "social linked data") is a proposed set of conventions and tools for building decentralized social applications based on Linked Data principles.

On the better web Berners-Lee envisions, users control where their data is stored and how it's accessed: social networks would still run in the cloud, but users could store their data locally, or they could choose a different cloud server run by a company or community they trust[91].

Nevertheless, technical feasibility alone does not guarantee the sort of widespread adoption necessary to build a useful platform. Some of the more mature tools developed in this space have faced serious difficulties in attracting a permanent user base, and the problems those platforms suffer from may hinder the growth of new systems as well. Social networks, in particular, are difficult to bootstrap due to network effects. We generally join social networks because our friends are already there.

Taking a competitive, rather than complementary, position in the market creates a difficult barrier to entry for new projects. Similarly, interoperable protocols require adoption at the developer level. Solid, which hopes to bridge between existing and novel social networks, faces a serious adoption challenge: Why should developers choose to switch to Solid's new data model, and what's the incentive for Facebook to make their data interoperable without legal requirements forcing them to do so?[92]

Even though promising, since it's still based on the same protocols, and since it's mainly focused on social networks, it doesn't really solve all the problems the Web suffers from.

---

[90] D. Weinberger. (August 2016). *How the father of the World Wide Web plans to reclaim it from Facebook and Google.* (https://goo.gl/hkYxzT)
[91] MIT. *What is Solid?* (https://goo.gl/8tCvLK)
[92] Chelsea Barabas, Neha Narula, Ethan Zuckerman. (August 2017). *Defending Internet Freedom through Decentralization: Back to the Future?* (https://goo.gl/325M7p)

1.6.6 The Advertising Web

The advertising industry, a business whose very raison d'être is often at odds with the goals of people trying to achieve a task as quickly as possible, unsurprisingly, abused of the language: exploiting the possibility of opening browser windows dynamically, something that previously could only be done by the user manually.

A young developer named Ethan Zuckerman realised that he could spawn a new window with an advertisement in it. That allowed advertisers to put their message in front of website visitors. Not only that, but JavaScript could be used to spawn multiple windows, some of them visible, some of them hidden behind the current window. It was a fiendish solution.

Twenty years later, Zuckerman wrote[93]:

> *I wrote the code to launch the window and run an ad in it. I'm sorry.*

Pop-up (and pop-under) windows became so intolerable that browsers had to provide people with a means to block them.

However, the advertising industry later found other ways to abuse JavaScript. Ad-supported online publishers injected bloated and inefficient JavaScript into their pages, making them slow to load. JavaScript was also used to track people from site to site.

---

[93] E. Zuckerman. (August 2014). "*The internet's original sin*". (https://goo.gl/kFxeXP)

1.6.7 The Tracking Web

Every time you search using Google Search; see Google's Adsense adverts; or visit a site that uses Google's Analytics web statistics analysis, you are identifying yourself to Google.

Google can store your documents, e-mails and contacts for you, so you can conveniently access them from anywhere; and they can conveniently own great swathes of information about millions of people.

Google can track you because of the way the Web works. Your browser automatically sends certain information to a website whenever you click on a link or type in the address of a web page, that are: the IP address, which the website needs to reply to you, and the cookies.

Cookies are bits of information from a website, which are stored on your computer, and sent back to the website every time your browser connects to that site again. For example cookies often contain your password for a website, or the last time you visited it. That way, the website can identify you as a previous visitor and maybe personalise the site layout for you.

Your browser identifies you via cookies whenever it requests external elements on a page. Virtually all sites incorporate content not physically located on their own servers, such as pictures, music, videos, etc., as well as, Google maps, Google site search boxes, Google Adsense adverts. Every time they load, Google get their cookie, so they can know what sites you're reading.

Regardless of whether this could be considerate a legitimate trade off between privacy and personalization, this was not how the Web was intended for. The old "*nothing to hide, nothing to fear*" platitude should not be the right reaction.

1.6.8 The Fake Web

Berners-Lee added his voice to the chorus of people gravely concerned about the explosion of fake news.

Fake news consist in a deliberate misinformation designed to appeal to readers' biases and make money through viral clicks[94].

In an interview, Tim Berners-Lee said[95]:

> *Through the use of data science and armies of bots, those with bad intentions can game the system to spread misinformation for financial or political gain.*

In December, Facebook announced new steps to curb fake news on its platform after months of continuous criticism. The plan included new tools to make it easier for Facebook users to flag fake stories and a collaboration with the respected journalism organization the Poynter Institute to independently investigate claims.

In another interview, Tim Berners-Lee said[96]:

> *We must push back against misinformation by encouraging gatekeepers such as Google and Facebook to continue their efforts to combat the problem, while avoiding the creation of any central bodies to decide what is "true" or not.*

The real problem is that gatekeepers themselves, i.e. Google and Facebook, in the end, can be traced back to the cause of the problema, i.e. clickbait[97], for their remunerative advertising system[98].

---

[94] Helle Hunt. *What is fake news? How to spot it and what you can do to stop it.* (https://goo.gl/U3syy8)
[95] CBS News. Shanika Gunaratna (March 2017). *Web inventor Tim Berners-Lee on the biggest problems with the internet today.* (https://goo.gl/gxtjZb)
[96] Telegraph. *Sir Tim Berners-Lee, World Wide Web inventor, urges crackdown on 'shocking' fake news.* (https://goo.gl/5RFbzr)
[97] Wikipedia. *Clickbait.* (https://goo.gl/r6Y743)
[98] Google. *Discover how easy it is to use AdSense.* (https://goo.gl/SRSY4u)

Conclusions

The World Wide Web is enormous, but it very fragile. It has unified the information of the entire world, standardizing how we distribute and present information to each other, but the way content is distributed is fundamentally flawed.

The Web is centralized. When content is hyper-centralized, it makes us highly dependent on the Internet backbones to the datacenters functioning. Aside from making it easy for governments to block and censor content, there are also reliability problems. Even with redundancies, major backbones sometimes get damaged, or routing tables go haywire, and the consequences can be drastic.

While the Internet is a truly distributed system, designed so that if any one piece goes out, it will still function — the Web, built on top of the Internet, is not.

Evolving the Web infrastructure is nearly impossible, given the number of backwards compatibility constraints and the number of strong parties invested in the current model. However, new protocols have emerged, gaining wide use since the emergence of the Web, that can address and solve most of the problems the Web suffers from.

The content-addressing and the peer-to-peer networks.

The Web should be built over a content-addressed peer-to-peer network.

The next chapter is about the state-of-the-art content-addressed peer-to-peer networks.

CHAPTER 2
THE CONTENT-ADDRESSED PEER-TO-PEER NETWORKS

A distributed network, or peer-to-peer network, is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

2.1 The Peer-To-Peer Networks

2.1.1 Architecture

Distributed or peer-to-peer networks generally implement some form of virtual overlay network on top of the physical network topology, where the nodes in the overlay form a subset of the nodes in the physical network. Data is still exchanged directly over the underlying TCP/IP network, but at the application layer peers are able to communicate with each other directly, via the logical overlay links (each of which corresponds to a path through the underlying physical network). Overlays are used for indexing and peer discovery, and make the P2P system independent from the physical network topology.

The most successful distributed networks have been peer-to-peer file-sharing applications primarily geared toward large media (audio and video), such as Napster and BitTorrent.

2.1.2 File-sharing applications

Napster

Napster is a file-sharing computer service created by American college student Shawn Fanning in 1999. Napster allowed users to share, over the Internet, electronic copies of music stored on their personal computers.

The arrival of Napster in 1999 marked the emergence of decentralized peer-to- peer (P2P) sharing of music over the Internet. At its peak in 2001 there were as many as 1.5 million people simultaneously sharing files worldwide by using Napster. Napster had embedded in the consciousness of consumers the idea of downloading songs from the Internet — bypassing the purchase of established distribution forms, such as records, tapes, or compact discs (CDs).

The file sharing that resulted set in motion a legal battle over digital rights and the development of digital rights management software to prevent computer copyright piracy.

BitTorrent

BitTorrent is a protocol for sharing large computer files over the Internet. It was created in 2001 by Bram Cohen, an American computer programmer who was frustrated by the long download times that he experienced using applications such as FTP.

Files shared with BitTorrent are divided into smaller pieces for distribution among the protocol's users, called "peers". A peer who wishes to download a file is directed by a software application called a "client" to access a Web site that hosts a tracker. The tracker keeps records of all the peers who have previously downloaded the file and then allows pieces of their copies of the file to be downloaded by the peer conducting the search.

By breaking the file into smaller pieces and allowing peers to download those pieces from each other, BitTorrent uses much less bandwidth than would be the case if all the peers downloaded the complete file from the original source. Once a file is completely downloaded, it becomes a "seed", that is, a file from which other peers can download pieces. However, BitTorrent can also work without the existence of a seed; a group of peers can share pieces of a file as long as they have among them all the pieces of the original complete file. Some tracker Web sites encourage seeding by penalizing peers who do not seed their files after their downloads are complete.

When content is distributed among unknown peers, the problem of privacy arises.
Modern cryptography, in particular the asymmetric encryption based on public/private keys, offers a promising solution to the problem of the privacy in a peer-to-peer network Web.

2.1.3 Content-addressing

Content-addressing identifies a content by its “fingerprint” rather than its location. This fingerprint is a cryptographic hash of the content.

A cryptographic hash is a short string of letters and numbers that’s calculated by feeding the content into a cryptographic hash function like SHA.

The hash of a content uniquely identifies exactly that content. As long as the content stays the same, the hash of the content stays the same. If the content changes, even its hash changes. Two different files, with different filename but identical content, will be tracked with the same hash. Even though the files are distincts, if they have the same content, then they will produce (and will be identified by) the same hash.

This way to identify content, using the content’s cryptographic hash instead of the location, is called content-addressing.
The cryptographic hash for a piece of content never changes, which means content addressing guarantees that the links will always return the same content, regardless of where the content is retrieved from, regardless of who added the content to the network, and regardless of when the content was added.

The implications of content-addressing

A data structure that uses content-addressed links is a persistent data structure. The following are just a few implications of storing and sharing data using a content-addressed protocol[99].

Content-addressing increases the durability of data

It ensures that data will not become endangered as long as anyone is still relying on it because anyone can hold a valid copy of the data they care about. You will not have to worry about whether someone is going to turn off the servers where the data are hosted because you are one of the hosts. You and your peers hold the data among yourselves and are able to share the data directly with each other without relying on centralized points of failure.

Content-addressing increases the integrity of data

You can validate data by checking the data's fingerprints against the links. That kind of validation is impossible with location-addressed links. This is especially powerful on the large scale, where millions of websites and datasets reference each other billions of times. With location-addressed links, all of those connections are brittle. With content-addressed links, the connections become resilient and reliable.

Content-addressing allow content to be unavailable for a period

As soon as any node has the content, everyone's links start working. Even if someone destroys all the copies on the network, it only takes one node adding the content in order to restore availability. A cryptographic hash permanently points to the content it was derived from, so links permanently point to their content. Even if the content becomes unavailable for a period, the links will work as soon as anyone starts providing the content again.

Content-addressing is harder to attack and easier to recover

Even if the original publisher is taken down, the content can be served by anyone who has it. As long as at least one node on the network has a copy of the content, everyone will be able to get it. This means the responsibility for serving content can change over time without changing the way people link to the content and without any doubt that the content you're reading is exactly the content that was originally published.

[99] Matt Zumwalt. *Lesson: The Power of Content-addressing*. (https://goo.gl/ATvFkv)

2.2 The InterPlanetary File System

The InterPlanetary File System (IPFS) is a content-addressable, peer-to-peer hypermedia distribution protocol, to create a permanent and decentralized method of storing and sharing files[100].

IPFS is resistant to censorship and DDoS attacks, which HTTP struggles with: it presents an opportunity to construct a more resilient Web, whose links do not rot, whose files are deduplicated globally, with no single point of failure.

IPFS is a synthesis of well-tested internet technologies such as DHTs, the Git versioning system and Bittorrent, to let multiple nodes supply parts of a file all at the same time: if a web server goes down, it won't take all of the files on it with it.

In IPFS you separate the steps to find a content into two parts: identify the file with content addressing, and go and find it. When you have the hash then you ask the network you're connected to who has the content identified by the hash, and you connect to the corresponding nodes and download it.

[100] Juan Benet. (2014). *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)*. (https://goo.gl/zP7RcG)

2.2.1 IPFS network

IPFS can work in partitioned networks. You don't need a stable connection to the rest of the Web, in order to access content through IPFS. As long as you are connected to at least one node that can reach the content you want, you can access that content.

IPFS does not rely on DNS. If someone blocks your access to DNS or spoofs DNS in your network, it will not prevent IPFS nodes from resolving content over the peer-to-peer network.

IPFS does not rely on the Certificate Authority System, so bad or corrupt Certificate Authorities do not impact it.

IPFS nodes are all capable of serving the same content, so you're not stuck relying on one point of failure. IPFS nodes work hard to find each other on the network and to reconnect with each other after connections get cut.

With IPFS, people viewing the content are also helping distribute the content, unless they opt out, and anyone can choose to pin a copy of some content on their node in order to help with access and preservation.

Content can be moved via sneakernet. Even if your network is physically disconnected from the rest of the internet, you can write content from IPFS onto USB drives or other external drives, physically move them to computers connected to a new network, and re-publish the content on the new network.

Even though you're on a separate network, IPFS will let nodes access the content using the same identifiers in both networks as long as at least one node on the network has that content.

2.2.2 IPFS Objects

IPFS is essentially a peer-to-peer system for retrieving and sharing IPFS objects: it creates a peer-to-peer swarm that allows the exchange of IPFS objects.

An IPFS object is a data structure with two fields:

- Data - a blob of unstructured binary data.
- Links - an array of Link structures, that are links to other IPFS objects.

A Link structure has three data fields:

- Name - the name of the Link.
- Hash - the hash of the linked IPFS object.
- Size - the cumulative size of the linked IPFS object, including following its links; it's mainly used for optimizing the P2P networking.

IPFS objects are normally referred to by their Base58 encoded hash.

The totality of IPFS objects forms a cryptographically authenticated data structure, known as a Merkle DAG, that can be used to model many other data structures[101]. Merkel means that the structure is cryptographically verified, while DAG means that the structure is a Directed Acyclic Graph. In fact it is impossible to have cycles in this graph.

File System

IPFS can easily represent a file system consisting of files and directories.

A file is represented by an IPFS object with data being the file contents (plus a small header and footer) and no links, i.e. the links array is empty. Since the filename is not part of the IPFS object, two files with different names and the same content will have the same IPFS object representation and hence the same hash.

A directory is represented by a list of links to IPFS objects representing files or other directories. The names of the links are the names of the files and directories.

[101] Protocol Labs. *Technical specifications for the IPFS protocol stack.* (https://goo.gl/LvmS1h)

2.2.3 The InterPlanetary Linked Data

The InterPlanetary Linked Data (IPLD) is the data model of the content-addressable web for all hash-inspired protocols: it allows to traverse links across protocols, and to explore data regardless of the underlying protocol[102].

There are a variety of systems that use merkle-tree and hash-chain inspired data structures, such as git, bittorrent, tahoe-lafs, sfsro, and of course even IPFS.

IPLD (Inter Planetary Linked Data) defines:

- merkle-links: the core unit of a merkle-graph.
- merkle-dag: any graphs whose edges are merkle-links.
- merkle-paths: unix-style paths for traversing merkle-dags with named merkle-links
- a data model: a flexible, JSON-based data model for representing merkle-dags.
- a serialized formats: a set of formats in which IPLD objects can be represented, for example JSON, CBOR, CSON, YAML, Protobuf, XML, RDF, etc.
- a canonical format: a deterministic description on a serialized format that ensures the same logical object is always serialized to the exact same sequence of bits. This is critical for merkle-linking, and all cryptographic applications.

In short, IPLD is "JSON documents with named merkle-links that can be traversed".

[102] Protocol Labs. *IPLD Specification*. (https://goo.gl/uniHsv)

Merkle links

A merkle-link is a link between two objects which is content-addressed with the cryptographic hash of the target object, and embedded in the source object.

Content addressing with merkle-links allows (i) cryptographic integrity checking and (ii) immutable data-structures.
Resolving a link's value can be tested by hashing. In turn, this allows wide, secure, trustless exchanges of data, as others cannot give you any data that does not hash to the link's value.
Data structures with merkle links cannot mutate, which is a nice property for distributed systems. This is useful for versioning, for representing distributed mutable state (e.g. CRDTs), and for long term archival.
A merkle-link is represented in the IPLD object model by a map containing a single key `/` mapped to a "link value".

For example:

```
{
  "foo": {
    "bar": "/ipfs/QmUmg7BZC1YP1ca66rRtWKxpXp77WgVHrnv263JtDuvs2k",
    "baz": {
      "/": "/ipfs/QmUmg7BZC1YP1ca66rRtWKxpXp77WgVHrnv263JtDuvs2k"
    }
  }
}
```

Where:

- `foo/bar` is not a link
- `foo/baz` is a link
- `/` is the link key
- `/ipfs/QmUmg7BZC1YP1ca66rRtWKxpXp77WgVHrnv263JtDuvs2k` is the link value

Pesudo-link objects

A merkle link object cannot contain any other key besides "`/`".
However, it's possible to have pseudo link object.

For example:

```
{
  "files": {
    "cat.jpg": {
      "link": {
        "/": "/ipfs/QmUmg7BZC1YP1ca66rRtWKxpXp77WgVHrnv263JtDuvs2k"
      },
      "mode": 0755,
      "owner": "jbenet"
    }
  }
}
```

Where:

- `files/cat.jpg` is a pseudo "link object":
  it is not a link but it contains a link and other properties
- `files/cat.jpg/link` is the link

When dereferencing the link, the map itself is to be replaced by the object it points to unless the link path is invalid. The link can either be a multihash, in which case it is assumed that it is a link in the `/ipfs` hierarchy, or directly the absolute path to the object. Currently, only the `/ipfs` hierarchy is allowed.

If an application wants to use objects with a single `/` key for other purposes, the application itself is responsible to escape the `/` key in the IPLD object so that the application keys do not conflict with IPLD's special `/` key.

Merkle DAG

Objects with merkle-links form a graph, which necessarily is both directed, and which can be counted on to be Acyclic, if the properties of the cryptographic hash function hold, a merkle-dag. Hence all graphs which use merkle-linking (merkle-graph) are necessarily also Directed Acyclic Graphs (DAGs, hence Merkle-DAG).

Merkle paths

A merkle-path is a unix-style path (e.g. `/a/b/c/d`) which initially dereferences through a merkle-link and allows access of elements of the referenced node and other nodes transitively.

General purpose filesystems are encouraged to design an object model on top of IPLD that would be specialized for file manipulation and have specific path algorithms to query this model.

A merkle-path is a unix-style path which initially dereferences through a merkle-link and then follows named merkle-links in the intermediate objects. Following a name means looking into the object, finding the name and resolving the associated merkle-link.

For example:

```
/ipfs/QmUmg7BZC1YP1ca66rRtWKxpXp77WgVHrnv263JtDuvs2k/a/b/c/d
```

Where:

- `ipfs` is a protocol namespace
- `QmUmg7BZC1YP1ca66rRtWKxpXp77WgVHrnv263JtDuvs2k` is a hash
- `a/b/c/d` is a path traversal, as in unix

Path traversals, denoted with `/`, happen over two kinds of links:

- in-object traversals traverse data within the same object
- cross-object traversals traverse from one object to another

Data model

At its core, IPLD data-model “is just JSON” in that:

- it is also tree based documents with a few primitive types – it maps 1:1 to json,
- it can be used through JSON itself.

But, IPLD data-model “is not JSON” in that:

- it improves on some mistakes,
- it has an efficient serialized representation,
- it does not actually specify a single on-wire format, as the world is known to improve.

2.2.4 The InterPlanetary Naming System

IPFS hashes represent immutable data, which means they cannot be changed without the content being different. This is a good thing because it encourages data persistence, but it's also desiderable to find the latest IPFS hash representing the last version of a content. IPFS accomplishes this using the InterPlanetary Naming System (IPNS).

IPNS is a way to add a small amount of mutability to the permanent immutability of IPFS. IPNS allows you to store a reference to an IPFS hash under the namespace of the IPFS node you control.

Human-readable mutable addressing

IPFS/IPNS hashes are long strings that aren't easy to memorize. So IPFS allows you to use the existing Domain Name System (DNS) to provide human-readable links to IPFS/IPNS content. It does this by allowing you to insert the hash into a TXT record on your nameserver.

Going forward, IPFS has plans to also support Namecoin, which could theoretically be used to create a completely decentralized, distributed web that has no requirements for a central authority in the entire chain. No ICANN, no central servers, no politics, no expensive certificate "authorities", and no choke points.

In IPFS a mutable IPNS address is resolved to its corresponding IPFS address.

For example:

```
/ipns/example.com/foo/bar/baz.png
/ipfs/QmW98pJrc6FZ6/foo/bar/baz.png
```

Where:

- `/ipns/example.com` is the IPNS address
- `/ipfs/QmW98pJrc6FZ6` is the corresponding IPFS address
- `/foo/bar/baz.png` is the path to the resource.

The IPFS/HTTP gateway

The IPFS implementation ships with an HTTP gateway. The IPFS/HTTP gateway allows current web browsers to access IPFS until the browsers implement IPFS directly. It represents the bridge between the old and the new Web.

2.3 Dat

Dat is a new peer-to-peer hypermedia protocol. It provides public-key-addressed file archives which can be synced securely and browsed on-demand.

Dat archives sync from multiple sources at once, and result to be:

- secure, since all updates are signed and integrity-checked;
- resilient, since archives can change hosts without changing their URLs;
- versioned, since changes are written to an append-only version log;
- decentralized, since any device can host any archive.

Dat is a protocol designed for syncing folders of data, even if they are large or changing constantly. Dat uses a cryptographically secure register of changes to prove that the requested data version is distributed. A group of clients can connect to each other to form a public or private network to exchange data between each other[103].

Dat is free software built for the public by Code for Science & Society, a nonprofit. Researchers, analysts, libraries, and universities are already using Dat to archive and distribute scientific data[104]. Developers are building applications on Dat for browsing peer-to-peer websites and offline editable maps[105].

Usually moving large files and folders to other computers involves one of a few strategies:, such as being in the same location (usb stick), using a cloud service (Dropbox), or using old but reliable technical tools (rsync). However, none of these easily store, track, and share your data securely over time, and users often are stuck choosing between security, speed, or ease of use. Dat provides all three by using a state of the art technical foundation and user friendly tools for fast and secure file sharing that you control.

---

[103] K. M. Maxwell Ogden, M. B. Madsen, and Code for Science. (May 2017). *Dat - Distributed Dataset Synchronization And Versioning*. (https://goo.gl/CZDPm9)
[104] The New York Times. Amy Harmon (March 2017). *Activists Rush to Save Government Science Data — If They Can Find It*. (https://goo.gl/8nt4Xw)
[105] Aliya Ryan (March 2017). *Mapping Waorani Territory*. (https://goo.gl/uJcwqP)

2.3.1 Beaker Browser

Beaker is a new kind of browser that gives you the power to create websites, share files, and control your data[106]. It is a peer-to-peer browser with tools to create and host websites over DAT. It enhance the HTML5 API with the DatArchive API.

Website manifest

Websites and applications served over `dat://` can include a manifest file to specify metadata and configure special behaviors. The metadata file must be located at `/dat.json` in the root of the website[107]. Beaker automatically creates and manages the manifest for Dat archives created with the DatArchive Web API.

For example:

```
{
  "url": "dat://4483a2..66/",
  "title": "My website",
  "description": "A simple website built with the Beaker Browser",
  "fallback_page": "/public/404.html",
  "web_root": "/public"
}
```

Where:

- `url` is the Dat archive's URL.
- `title` is a short and descriptive human-friendly title.
- `description` is a description of the Dat archive.
- `repository` is the URL for the git repository associated with the Dat archive.
- `fallback_page` it the path to a fallback page to serve instead of the default 404 page.
- `web_root` it the path of the folder from which all requests should be served.

---

[106] Paul Frazee (December 2016). *Beaker: An Experimental P2P Browser*. (https://goo.gl/5Ly2D9)
[107] Paul Frazee. *dat.json site manifest*. (https://goo.gl/H3X9re)

DatArchive API

The DatArchive API is the Beaker's interface for reading and writing the Dat peer-to-peer filesystem. Websites and applications can use the DatArchive API to create, write, and read Dat archives[108].

By default, any `dat://` website or application can read other `dat://` pages with HTML embeds, Ajax, or the DatArchive read interfaces.

By default, `dat://` pages are granted permission to write to other `dat://` pages that it created. The user will be prompted to grant permission when a `dat://` page attempts to create a new Dat archive, or modify a `dat://` page that was created by a different `dat://` origin. Additionally, the user must be the owner of a given Dat archive in order to modify it.

The `dat.json` file is a special file that specifies metadata and configuration. It cannot be written directly using the DatArchive API.

How Beaker manages a Dat archive's disk usage depends on whether the user owns the Dat archive, has chosen to seed the archive's files, or is simply visiting and browsing the archive's files. If the user owns a given Dat archive or has chosen to seed its files, Beaker keeps those files on the local disk. All other `dat://` websites, applications, and files are kept on the local disk temporarily, and will be automatically deleted. Files that are temporarily cached after visiting a `dat://` page can also be manually purged in the Beaker settings page.

[108] Paul Frazee. *DatArchive API*. (https://goo.gl/SQdQfP)

WebDB

WebDB is a database that reads and writes records on `dat://` websites. It abstracts over the DatArchive API to provide a simple database-like interface, inspired by Dexie.js[109], and leveraging on the HTML5 IndexedDB API[110] using the wrapper level.js[111].

WebDB scans a set of source Dat archives for files that match a path pattern: it caches and indexes those files so they can be queried easily and quickly. WebDB also provides a simple interface for adding, editing, and removing records from archives.

WebDB sits on top of Dat archives. It duplicates ingested data into IndexedDB, which acts as a throwaway cache. The cached data can be reconstructed at any time from the source Dat archives.

WebDB treats individual files in the Dat archive as individual records in a table. As a result, there's a direct mapping for each table to a folder of JSON files. This is for performance and linkability: when a record is created, peers in the network will only download the newly-created file, instead of re-download the entire file; furthermore putting each record in an individual file also makes each record linkable.

For example, a "posts" table might map to the `/posts/*.json` files.
WebDB's mutators, like `put()`, `add()` and `update()`, simply write records as JSON files in the `posts/` directory; while WebDB's readers and query-ers, like `get()` and `where()`, simply read from the IndexedDB cache.

WebDB watches its source archives for changes to the JSON files that compose its records. When the files change, it syncs and reads the changes, then updates IndexedDB, keeping query results up-to-date.

---

[109] David Fahlander. *A Minimalistic Wrapper for IndexedDB.* (https://goo.gl/zT8hnU)
[110] W3C. *Indexed Database API 3.0* (https://goo.gl/29BJZA)
[111] Max Ogden. *An abstract-leveldown compliant store on top of IndexedDB.* (https://goo.gl/6waAWj)

2.3.2 HashBase

Hashbase is a public peer for files published with the Dat protocol[112].

Publishing with Dat means that peers will contribute bandwidth, but only if they're online and sharing your files. If nobody's hosting your files, then they won't be accessible. Hashbase acts as a "super peer" that makes sure your content is always available, so your files are always available, even when you're offline.

[112] Paul Frazee. (July 2017). *Introducing Hashbase* (https://goo.gl/VEFBDB)

2.4 ZeroNet

ZeroNet is a tool to build and deploy websites over the BitTorrent peer-to-peer network, using the modern cryptography[113], to let identify the website and to allow only the owner to publish changes[114].

2.4.1 Using ZeroNet

Create a site

When you create a site, ZeroNet generates two keys: a private key and a public key.

The private key allows you to sign new content for your site: it is impossible to modify your site without it. It is private: it is not stored in a central registry, you only have it.

The public key is the site address: it allows anyone to verify if the site is not altered by others. Every downloaded file is verified: this makes it safe from any malicious modification.

ZeroNet uses the same elliptic curve based encryption as in Bitcoin: using the current fastest supercomputer, it would take around 1 billion years to "hack" a private key.

Visit a site

When you visit a site, ZeroNet gathers your IP address to register you as a visitor, and downloads a file named `content.json`, that describe all the site.

The `content.json` holds all other file-names, hashes and the site owner's cryptographic signature, that is used to verify the files of the site.

Visitors start serving sites as soon as they visit them. You can use the Tor network to hide your real IP address.

[113] T. Kysar. (March 2015). *ZeroNet Expands Key Distributed And Anonymous Features.* (https://goo.gl/3Hcktz)
[114] Andy. (March 2016). *Play: A P2P Distributed Torrent Site That's Impossible to Shut Down.* (https://goo.gl/iL72AL)

Update a site

When you update your site, ZeroNet signs the new `content.json` using your keys, and sends the new `content.json` to the other visitors. Visitors check if the file is newer than their current file, and if so, download the changed files.

The browser is notified immediately about the file changes using the WebSocket API[115]: this allows real-time updated sites.

Update a site of another user

If you want to update the site of another user, you can request permission to edit from the site owner by sending your auth address to the site owner. If your request to edit is accepted, the site owner creates a new file and set your auth address as the valid signer, and publishes the new file and the changed permission to other visitors.

The site owner is able to remove misbehaving users.
The user files size can be limited to help avoid spamming.
An unique, BIP32[116] bases, valid BitCoin address is generated for every user of the site.

[115] WhatWG (June 2018). *HTML Living Standard - Web sockets*. (https://goo.gl/LS29S4)
[116] Bitcoin. *BIP 32 Specification: Key derivation*. (https://goo.gl/ds4RiZ)

2.5 Modern Cryptography

Modern cryptography was invented in the 1970s and is a mathematical foundation for computer and information security. Since its invention, several suitable mathematical functions, such as prime number exponentiation and elliptic curve multiplication, have been discovered. These mathematical functions are practically irreversible, meaning that they are easy to calculate in one direction and infeasible to calculate in the opposite direction. Based on these mathematical functions, cryptography enables the creation of digital secrets and unforgeable digital signatures.

2.5.1 Private and Public Keys

An identity in a peer-to-peer network is a key pair, consisting of a private key and a public key. The private key k is a number, usually picked at random.

From the private key, we can use elliptic curve multiplication, a one-way cryptographic function, to generate a public key K.
From the public key K, we can use a one-way cryptographic hash function to generate a resource address A.

There is a mathematical relationship between the public and the private key that allows the private key to be used to generate signatures on messages. This signature can be validated against the public key without revealing the private key.

Private Keys

A private key is simply a number, picked at random. Ownership and control over the private key is the root of user control over all funds associated with the corresponding bitcoin address. The private key must be backed up and protected from accidental loss, because if it's lost it cannot be recovered and the data secured by it are forever lost, too.

You can pick your private keys randomly using just a coin, pencil, and paper: toss a coin 256 times and you have the binary digits of a random private key. The public key can then be generated from the private key.

Public Keys

The public key is calculated from the private key using elliptic curve multiplication, which is irreversible.

$$K = k\,G$$

Where:

- k is the private key
- G is a constant point called the generator point
- K is the resulting public key

The reverse operation, known as "finding the discrete logarithm", calculating k if you know K, is as difficult as trying all possible values of k (i.e., a brute-force search).

2.5.2 Key Formats

Both private and public keys can be represented in a number of different formats. These representations all encode the same number, even though they look different.
These formats are primarily used to make it easy for people to read and transcribe keys without introducing errors. However, these keys are very hard to remember.

Deterministic Key generation

A method for generating multiple keys is the deterministic key generation. Here you derive each new private key, using a one-way hash function from a previous private key, linking them in a sequence. As long as you can recreate that sequence, you only need the first key (known as a seed or master key) to generate them all. Here, the private keys are all derived from a common seed, through the use of a one-way hash function.

The seed is a randomly generated number that is combined with other data, such as an index number or "chain code" to derive the private keys. The seed is sufficient to recover all the derived keys, and therefore a single backup at creation time is sufficient.

CONCLUSIONS

The InterPlanetary File System can address and solve most of the problems the Web suffers from, and enable a truly distributed Web.

The next chapters goes back to the Web, exploring the systems at the state-of-the-art that allow to create websites for the Web, to know how we can use such systems, or improve them, to build and deliver websites over a peer-to-peer network.

In particular, the next chapter, introduces the Web Content Management Systems.

CHAPTER 3
WEBSITE DESIGN AND THE CONTENT MANAGEMENT SYSTEMS

In this chapter the Web Content Management Systems are introduced.
In particular, the traditional Web Content Management Systems; the Static Site Generators; the Decoupled and Headless CMSs; and the Static Progressive Web App Generators.

3.1 Websites Design

3.1.1 Static Websites

The Web is an exponentially growing set of interlinked pages. The indexed Web, which is the portion of the Web that can be retrieved by a Web search engine, contains around 4.5 billion pages[117]. The web page is the unit of measurement.

Web pages are grouped together to form websites. Designing a website involves several facets. Most importantly you need to identify the purpose of the website, the audience you target to, and the appropriate content, then to be aware of the technical considerations involved in displaying a web page, looking at other website with a similar purpose and coming up with an appropriate design.

Designing a website requires to come up with a scheme for presenting and arranging the information. The presentation ought to support the fact that the information is related, and the arrangement often should mirror the logical structure of the information.

A Web page could be written using a simple text editor, but there are other types of editors or ways to create a Web page. These include HTML editors—text editors that include easy ways to insert HTML tags into a file, and visual editors—editors that provide tools to produce a Web page without you having to insert raw HTML.

Putting a website on the Web means taking the source files and placing them on a Web server. Some Internet service providers, Web-based services, and organizations provide space for web pages.

[117] WorldWideWebSize.org. (June 2018). *The size of the World Wide Web (The Internet)*. (https://goo.gl/94pFJP)

Website pages

In general, a website presents two kind of pages:

- an “index page”, or list page, that shows a list of entries in a summarized way
- a “single page”, item page, that shows a particular item in a detailed way

For example, university sites show information about their campuses, faculties for each campus, courses for each faculty; news sites show the latest news stories; personal blog sites, show personal observations or reviews.

In the following, we consider Blog websites. In a Blog, content range from the personal to the political, and can focus on one narrow subject or a whole range of subjects. In the latter case, content is categorized by means tags.

Why the name Blog

“Blog” is an abbreviated version of “weblog” (web-log), a term used to describe an ongoing chronicle of information published onto the web. A Blog is website that maintains an ongoing chronicle of information. A blog features diary-type commentary and links to articles on other websites, usually presented as a list of entries, also known as “posts”, in reverse chronological order. A blogs have multiple authors, each writing his/her own articles.[118]

[118] WordPress.org. *Introduction to Blogging*. (https://goo.gl/31spsJ)

Index page

For example, in `index.html`:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Blog</title>
</head>
<body>
  <header>
    <h1>My Blog</h1>
    <h2>Articles</h2>
  <header>
  <section>
    <p>There are <strong>2</strong> articles:</p>
    <ul>
      <li><a href="/post-1.html">My first post</a></li>
      <li><a href="/post-2.html">My second post</a></li>
    </ul>
  </section>
</body>
</html>
```

We note that, every time a new post is added, the `index.html` code must be updated.

Single pages

For example, in `/post-1.html`:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Blog</title>
</head>
<body>
  <header>
    <h1>My first post</h1>
  <header>
  <section>
    <p>Hello static websites!</p>
  </section>
</body>
</html>
```

For example, in `/post-2.html`:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Blog</title>
</head>
<body>
  <header>
    <h1>My second post</h1>
  <header>
  <section>
    <p>Bye static websites!</p>
  </section>
</body>
</html>
```

We note that, single pages share the same page structures: they differ just for their content.

3.1.2 Dynamic websites

It is pleonastic that handwriting web pages is not suitable to design websites. Since the dawn of the Web, the need to automate the production of webpages emerged. That is: programs or scripts that resolve common tasks in web development and ease the production of webpages.

The most used server-side scripting language, that is specially suited to web development, to dynamically render web pages, is PHP.

PHP

PHP is open-source, free to download[119] and use: it runs on various platforms (such as Windows, Linux, Unix, Mac OS X, etc.) and it is compatible with almost all servers used today (such as Apache, IIS, etc.).

At the time of writing, PHP is used by over 83% of all the websites whose server-side programming language is known[120], running some of the largest websites on the planet, such as Facebook and Wikipedia.org[121].

Why the name PHP

PHP is the acronym of the original definition "*Personal Home Page/Forms Interpreter*", or for the "recursive" definition "*PHP: Hypertext Preprocessor*".

---

[119] PHP.net. *The official PHP resource website*. (https://goo.gl/zz9umJ)
[120] W3Tech. (June, 2018). *Usage statistics and market share of PHP for websites*. (https://goo.gl/1DeRxi)
[121] W3Techs. *Usage statistics and market share of PHP for websites*. (https://goo.gl/WZfxeH)

Brief history of PHP

PHP development began in 1994 when Rasmus Lerdorf wrote several Common Gateway Interface (CGI) programs in C, which he used to maintain his personal homepage. He extended them to work with web forms and to communicate with databases. That implementation would become PHP.

Rasmus Lerdorf believes there are four kinds of programmers: the first, the pragmatic ones who are just after solving their own problems; the second kind finds programming as a means of self-expression; the third are the real programmers who enjoy programming for its own sake; the fourth are the open source zealots who wish to change the world.
He claims to be of the first kind. He programs to solve his problem and then moves on. He confesses that he created PHP purely to serve his own interest, to solve his own set of problems. Then, he made the source publicly available so others could benefit from it.

That decreted the success of PHP.

Using PHP

PHP is mainly used to generate dynamic page content: its code is executed on the server and produces plain HTML.

PHP can use the file system, to create, open, read, write, delete, and close files.
PHP can connect to and manipulate database: it can add, delete, modify data in database.
PHP supports a wide range of databases, such as MySQL.

MySQL is the most popular SQL database system used on the web with PHP.
MySQL is developed, distributed, and supported by Oracle Corporation, but is free to download and use[122], on various platforms (Windows, Linux, Unix, Mac OS X, etc.).

---

[122] MySQL.com. *MySQL Downloads*. (https://goo.gl/xV5g7Z)

Index page

For example, the index page could be something like the following:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Blog</title>
</head>
<body>
<header>
  <h1><?php echo $blog->title ?></h1>
  <h2>Articles</h2>
<header>
<section>
  <?php $articles = $database.find($query_articles) ?>
  <p>There are <strong><?php echo count($articles) ?></strong> articles:</p>
  <ul>
    <?php foreach ($articles as $article): ?>
      <li><a href="<?php echo $article->path ?>">
        <?php echo $article->title ?>
      </a></li>
    <?php endforeach ?>
  </ul>
</section>
</body>
</html>
```

Single page

For example, the single page could be something like the following:

```
<!DOCTYPE html>
<html>
<head>
  <title><?php $blog->title ?></title>
</head>
<body>
  <?php $article = $database.find($query_article) ?>
  <header>
    <h1><?php $article->title ?></h1>
  <header>
  <section>
    <?php $article->content ?>
  </section>
</body>
</html>
```

We note that, every time a new post is added, or an existing post, you do not need to add or update any page, because the HTML pages are dynamically generated, server-side rendered.

Routing

We also note that, dynamic websites, introduce the need for a new component, i.e. the *router*, which takes request for a page and select the right template to be rendered to dynamically generate the page.

3.2 The Content Management Systems

The introduction of PHP has facilitated the development of dynamic websites, but it has not resolved the hosting and publishing problem.

In the article "*Putting Information onto the Web*"[123], preserved since 1995 for archival and historical interest (as it is remarkably stated in the webpage), the W3C shows the ways to publish on the Web:

> *If you would like to create information and place it on the World Wide Web, you can approach this in several different ways [...]*
> *As an author, you will need to know how to create and edit hypertext [...]*
> *As a webmaster, you will probably want to learn how to organize large amounts of hypertext [...]*
> *As a system administrator, you will need to know the actual installation and configuration of a Web Server, and the use of things like CGI, forms, databases, and applications which will work together on your system [...]*

Web Server, *database*, *applications*: it is clear that publishing on the Web is not accessible to all a priori.

Since the beginning of Web development, the need to simplify the Web publishing process made the success of systems that support the management of the content of websites and the delivery onto the Web: the Web Content Management Systems (the Web CMSs).

Nowadays, there are different CMSs that can be used, but few of them have become clearly predominant over the rest of the competition. With over the 50% of the CMS market share, WordPress is the most used Web CMS, followed by Joomla and Drupal, both with less than 10% of the CMS market share[124].

[123] W3C (April 1995). *Putting information into the Web*. (https://goo.gl/uaKd4G)
[124] W3Techs. (June, 2018). *Usage of content management systems for websites*. (https://goo.gl/LUfh5p)

3.2.1 WordPress

WordPress is the most used Web CMS. It is used by over 31% of all websites in the Web (over 15M websites), that is: every three website you visit, one is made by WordPress.

WordPress was designed by Matt Mullenweg and released in 2003[125].
Since its release, in just a decade, WordPress would become the most used CMS in the Web.

How WordPress works

The main process of WordPress is called “the Loop”[126].
In general, WordPress:

- verifies that all the files it needs are present;
- collects the default settings, e.g. the number of posts per page, from the database;
- checks the user request to determine which posts to fetch from the database;
- retrieves the specified information from the database;
- stores the results in a variable used to render the appropriate template.

In synthesis, when a user visits a page, WordPress loads a template based on the request, parses the PHP in the template and returns the HTML to the user.

The type of content that is displayed in by the template file is determined by the Post Type[127] associated with the template file, and by the Template Hierarchy.

The Post Type describes the type of content. The default post type are: post, page, and attachment. Post types can be created by means Custom Post types.

The Template Hierarchy describes which template file WordPress will load based on the type of request and whether the template exists in the theme.

---

[125] M. Mullenweg. (May 2003). WordPress Now Available. (https://goo.gl/whcSjD)
[126] WordPress.org. *The Loop*. (https://goo.gl/9WSjWy)
[127] WordPress.org. *Post Types*. (https://goo.gl/L1Gwgr)

The Architecture of WordPress

WordPress consists of three major components:

- core
- themes
- plugins

The core of WordPress is designed to be lean and lightweight to maximize flexibility.

Plugins are ways to extend the core.
Plugins control the behavior and add functionality to the site.

Themes are ways to affect the overall look and feel of a site.
Themes control the presentation of content.

WordPress owes its success to the theme-and-plugin ecosystem.
Any software platform that aims to be widely used should provide a way to be customized, extended, and to allow the community of developers to contribute to the rise of the platform.
Any Web Content Management System that aims to be widely used should provide a themes-and-plugins ecosystem.

WordPress ecosystem

A flourishing software ecosystem couldn't exist without a fervent community behind it.

WordPress is available in over 160 languages around the world; its plugin authors are located around the globe. In 2014, there were 80 official WordCamps held in 29 countries.
There are over 840 meetup groups for WordPress all over the world, and over 249,000 active members in WordPress meetup groups all over the world, in 66 countries and 535 cities. The official WordPress support forum counts over 2 million total topics.

WordPress users often turn to developer created premium themes and plugins to customize and enhance their sites. These thriving marketplaces have actually sparked an entirely new economy within the WordPress ecosystem, which offers numerous opportunities for theme and plugin developers around the world.

Plugins marketplace

Themes and plugins are two of the most profitable businesses in the WordPress workspace. The industry has experienced a great deal of growth.
Since 2010, the plugin marketplace has grown over 3,000%.
In 2014, WordPress.org counted over 1B downloads of plugins.

There are currently more than 40,000 plugins available on WordPress.org, which have been collectively downloaded more than 1.2 billion times. In particular: 19 plugins have reached over 1 Million downloads; 11 plugins have reached over 7 Million downloads.
In CodeCanyon, a plugins marketplace (for WordPress and other CMSs), 80% of searches are focused on functionality and interface elements (e.g. sliders, forms, calendars, etc.).

Plugins claim over $70 million in total revenue. Utility plugins and interface elements are the two most profitable plugin types, each bringing in over $7 million. The combined $14 million for the top two categories starkly contrasts with the bottom three revenue types: plugins for SEO, forums, and auctions. These revenue streams equal less than $1 million combined. Social networking ranked No. 8 out of 17 types of plugins.

Themes marketplace

There are thousands upon thousands of themes to choose from with WordPress.
While many of the themes are available for free download, the more elaborate and skillful premium designs come at a cost. WordPress.org recommends over 80 different companies that provide themes, which are just the tip of the iceberg.
In 2014, WordPress.org counted over 123M downloads of themes.

The market for themes is larger than the market for plugins, at least for now.
Themes claim over $232 million in total revenue. According to Alexa.com, ThemeForest is one of the most popular 500 websites on the internet worldwide. Actually, the site boasts over 6,300 WordPress themes available for purchase and download.

The theme categories are as varied as the plugins. However, the top two categories, corporate and creative themes, account for over $125 million in revenue. That’s over 50% of the total revenue of themes.

3.2.2 Templates

WordPress can use different template files for displaying the website in different ways.
In the default WordPress theme, there are template files for the index view, category view, and archive view, as well as a template for viewing individual posts.

Index template

The index page is stored in the `index.php` file.

```
<?php get_header() ?>
<?php if (have_posts()) : ?>
  <?php while (have_posts()) : the_post() ?>
    <div class="post">
      <h2><a href="<?php the_permalink() ?>"><?php the_title() ?></a></h2>
      <small><?php the_time('F jS, Y') ?> by <?php the_author() ?></small>
      <p class="postmetadata">Posted in <?php the_category(', ') ?></p>
      <div class="entry"><?php the_content('Read more;'); ?></div>
    </div>
  <?php endwhile ?>
  <div class="navigation">
    <?php posts_nav_link('','','« Previous Entries') ?>
    <?php posts_nav_link('','Next Entries »','') ?>
  </div>
<?php else : ?>
  <h2>Sorry! Page Not Found</h2>
<?php endif ?>
<?php get_sidebar() ?>
<?php get_footer() ?>
```

Where:

- `have_posts()` checks for a next item in the current collection of posts.
- `the_post()` retrieves that next and makes it available for use.
- `the_content()` fetches the content of the post, filters it[128], and then displays it: it accepts a string, that is used for the “Read More” link after the excerpt[129].
- `posts_nav_link()` displays navigation controls to move forward/backward.

Also, where:

- `get_header()`, `get_sidebar()`, `get_footer()` are Include Tags.

[128] WordPress.org. *Plugin API/Filter Reference*. (https://goo.gl/THApRA)
[129] WordPress.org. *Customizing the Read More*. (https://goo.gl/Ljh3rR)

Archive template

An archive is a collection of historical posts.

If a visitor requests for a specific year, month, or date, WordPress will prepare the Loop with posts from that year, month or date only, using the archive template.

For example:

- `http://example.org/blog/index.php?y=2018`
- `http://example.org/blog/index.php?m=2018-09`
- `http://example.org/blog/2018`
- `http://example.org/blog/2018/09`

When WordPress prepares an archive view, it looks for a file named `archive.php`.
If that file does not exist, for the template hierarchy, it use `index.php`.
To visually disambiguate archives from the front page, create the `archive.php`[130].

Category template

If a visitor requests for a specific category, WordPress will prepare The Loop with posts from that category only, using the category template.

For example:

- `http://example.org/blog/index.php?c=category_name`

When WordPress prepares a category view, it looks for a file named `category.php`.
If that file does not exist, for the template hierarchy, it use `index.php`.
To visually disambiguate category from the front page, create the `category.php`.

It is possible to create separate template files for each category. Simply name the template file `category-X.php`, where `X` is the numerical ID of the category.

---

[130] WordPress.org. *Creating an Archive Index.* (https://goo.gl/xbTCXz)

Template Tags

To retrieve data from the database and print dynamic content into templates, WordPress offers the template tags.

A template tag is simply a piece of code that contains PHP function with optional parameters, that is used to print dynamic content, from the title of the page to an entire post.

For example:

- `the_title()` gets the title of the page.
- `the_post()` gets the current post.
- `bloginfo("name")` gets the blog name.
- `bloginfo("version")` gets the version of WordPress the website is running on.

Template partials

A template partial is a piece of a template that is included as a part of another template, such as a site header. Template partials can be embedded in multiple templates, simplifying theme creation.

Common template partials include:

- `header.php` for generating the site's header
- `sidebar.php` for generating the sidebar
- `footer.php` for generating the footer

You can create any number of template partials and include them in other template files.

Include Tags

To allow you to define a standard header, sidebar, footer for the website, WordPress offers the Include Tags[131].

An include tag is simply a piece of code that dynamically include other PHP files into the template. Any changes made to these files will immediately be made visible.

For example:

- `get_header()` prints the `header.php` file.
- `get_sidebar()` prints the `sidebar.php` file.
- `get_footer()` prints the `footer.php` file.

---

[131] WordPress.org. *Include Tags*. (https://goo.gl/wugQSU)

Template Hierarchy

To decide which template or set of templates should be used to display the page, WordPress uses the query string of the URL, and selects the template in the order determined by a template hierarchy[132].

Since templates are hierarchically ordered, if WordPress cannot find a template file with a matching name, it will skip to the next file in the hierarchy. If WordPress cannot find any matching template file, it will use the index.php file.

For example, if a visitor requests for the page of category page, let it be A with ID 1, WordPress will look in order for the following template files:

- a template file that matches the category slug, i.e. `category-A.php`.
- a template file that matches the category ID, i.e. `category-1.php`.
- a generic category template file, i.e. `category.php`.
- a generic archive template file, i.e. `archive.php`.
- the main theme template file, i.e. `index.php`.

[132] WordPress.org. *Template Hierarchy*. (https://goo.gl/tLGTxd)

3.2.3 Post Types

There are many different types of content in WordPress. These content types are normally described as post types.

The default post types are: Post, Page, Attachment.
In addition to the default post types, you can also create Custom Post types.

The default post types can be modified and removed by a plugin or theme, but it is not recommended that you remove built-in functionality for a widely-distributed theme or plugin.

Posts

Posts are used in blogs:

- posts are displayed in reverse sequential order by time, with the newest post first;
- posts have a date and time stamp;
- posts may have the default taxonomies of categories and tags[133] applied;
- posts are used for creating feeds.

The template files that display the Post post type are: `single.php` and `single-post.php`; `category.php` and all its iterations; `tag.php` and all its iterations; `taxonomy.php` and all its iterations; `archive.php` and all its iterations; `author.php` and all its iterations; `date.php` and all its iterations; `search.php`; `home.php` and `index.php`.

Pages

Pages are a static post type, outside of the normal blog stream/feed:

- pages are non-time dependent and without a time stamp
- pages are not organized using the categories and/or tags taxonomies
- pages can have page templates applied to them
- pages can be organized in a hierarchical structure

The template files that display the Page post type are: `page.php` and all its iterations; `custom.php` and all its iterations; `front-page.php`; `search.php` and `index.php`.

[133] WordPress.org. *Categories, Tags, & Custom Taxonomies.* (https://goo.gl/wKXsbB)

Attachments

Attachments are commonly used to display images or media in content, and may also be used to link to relevant files. They contain information about files uploaded through the media upload system, such as name and description, or, for images, such as size, location, etc.

The template files that display the Attachment post type are in order: {mime-type}_type.php; attachment.php; single-attachment.php; single.php and index.php

Custom Post Types

Custom Posts allow you to create your own post type.

The template files that display the Custom Post Type are:
`single-{post-type}.php`; `archive-{post-type}.php`; `search.php`; `index.php`.

While you generally won't develop Custom Post Types in your theme, you may want to code ways to display Custom Post Types that were created by a plugin. This ensures the portability of your user's content, and that if the theme is changed the content stored in the Custom Post Types won't disappear.

3.2.4 Plugins

Plugins are packages of code that extend the core functionality of the system.

Plugins allow you to extend the functionality of WordPress without touching WordPress core itself. If there's one cardinal rule in WordPress development, it's this: "Don't touch WordPress core". This means that you should not edit WordPress core files to add functionality to your site, because, when WordPress updates to a new version, it overwrites all the core files. Instead, to add functionality to your site, you should write a new plugin[134] or choose an existed one.

In WordPress, a plugin is a set of one or more PHP functions, that adds a specific set of features to the website.

You can seamlessly integrate a plugin with the site using access points and methods provided by the WordPress Plugin Application Program Interface (API)[135].

Plugin repositories

WordPress Plugins are available from several sources. The most popular and official source for WordPress Plugins is the WordPress.org repo.
The following two plugins[136] are included with WordPress core.

Akismet

Askimet is an advanced hosted anti-spam service[137].
It is the most popular plugin of all time: it has reached over 35 Million active install marks.
It is not surprising that such a service is needed in the WordPress platform.

Hello Dolly

"Hello Dolly" is the world's first official WordPress Plugin[138].
This is not just a plugin, it symbolizes the hope and enthusiasm of an entire generation summed up in two words sung most famously by Louis Armstrong: Hello, Dolly.
When enabled you will randomly see a lyric from "Hello, Dolly" in the Administration page.

[134] WordPress.org. *Writing a Plugin*. (https://goo.gl/ijr99c)
[135] WordPress.org. *Introduction to Plugin Development*. (https://goo.gl/igchR4)
[136] WordPress.org. *Plugins*. (https://goo.gl/6AoQGj)
[137] WordPress.org. *Akismet Anti-Span WordPress plugin*. (https://goo.gl/gUcLbw)
[138] WordPress.org. *Hello Dolly WordPress plugin*. (https://goo.gl/ZxXfmm)

Plugin Hooks

To allow plugins to “hook into” the main process, WordPress provided hooks.
While WordPress is running, at various times it checks if there are functions registered to run at that time: if so, then it execute these functions.

For example, before WordPress print the title of a post , it first checks if there are any function registered for the filter hook `the_title`: if so, the title text is passed in turn through each registered function and the final output is the end result of any and all of these registered functions.

Actions and Filters

WordPress provides two kinds of hooks: actions and filters.

Actions are functions triggered by specific events that take place in WordPress, such as publishing a post or changing themes.

Filters are functions that WordPress passes data through, at certain points in execution, just before taking some action with the data. Filters sit between the database and the browser (when WordPress is generating pages), and between the browser and the database (when WordPress is adding new posts to the database).

Sometimes the same goal can be accomplished with either an action or a filter.
For example, to change the text of a post, you might add a function to `publish_post` action hook (so the post is modified as it is saved to the database), or a function to `the_content` filter hook (so the post is modified as it is displayed in the browser screen).

3.2.5 Themes

A WordPress theme changes the design of your website, often including its layout. Changing your theme changes how your site looks on the front-end, i.e. what a visitor sees when they browse to your site on the web.

Themes take the content and data stored by WordPress and display it in the browser. When you create a WordPress theme, you decide how that content looks and is displayed. There are many options available to you when building your theme.

For example:

- Your theme can have different layouts, such as static or responsive.
- Your theme can display content anywhere you want it to be displayed.
- Your theme can specify which devices or actions make your content visible.
- Your theme can customize its typography and design elements using CSS.

WordPress themes are incredibly powerful. But, as with every web design project, a theme is more than color and layout. Good themes improve engagement with your website's content in addition to being beautiful.

At their most basic level, WordPress themes are collections of different files that work together to create what you see, as well as how your site behaves.

Required files

There are only two files absolutely required in a WordPress theme:

1. index.php – the main template file
2. style.css – the main style file

The most critical template file is index.php, which is the catch-all template if a more-specific template can not be found in the template hierarchy.

Although a theme only needs a index.php template, typically themes include numerous templates to display different content types and contexts, and additional files such as JavaScript files, CSS files, graphics and text files, and usually license.txt file, a `readme.txt` file, and a `changelog` file.

Common template files

The basic theme templates and files recognized by WordPress are:

```
.
├── assets
|   ├── js (dir)
|   ├── css (dir)
|   └── images (dir)
├── template-parts
|   ├── footer (dir)
|   ├── header (dir)
|   ├── navigation (dir)
|   ├── page (dir)
|   └── post (dir)
├── 404.php
├── archive.php
├── comments.php
├── footer.php
├── front-page.php
├── functions.php
├── header.php
├── index.php
├── page.php
├── README.txt
├── rtl.css
├── screenshot.png
├── search.php
├── searchform.php
├── sidebar.php
├── single.php
└── style.css
```

Where:

- `index.php` is the main template file.
- `style.css` is the main stylesheet.
- `rtl.css` is the right-to-left stylesheet, if the language's text direction is right-to-left.
- `front-page.php` is the front page template
- `home.php` is the home page template, that is the front page by default.
- `header.php` is the header template file, that is the <head> of the HTML document.
- `singular.php` is the singular template.
- `single.php` is the single post template is used when a visitor requests a single post.
- `single-{post-type}.php` is the single post template for a custom post type.
- `archive-{post-type}.php` is the archive post type for a custom post type.
- `page.php` is the page template for individual pages.
- `page-{slug}.php` is the page template for a specific slug e.g. "about".
- `category.php` is the category template to show posts by category.
- `tag.php` is the tag template to show posts by tag.
- `taxonomy.php` is the taxonomy term template to show a term in a custom taxonomy.
- `author.php` is the author page template.
- `date.php` is the date/time template to show posts by date or time.
- `archive.php` is the archive template to show posts by category, author, or date.
- `search.php` is the search results template to show users search results.
- `attachment.php` is the attachment template to show a single attachment.
- `image.php` is the image attachment template for a single image attachment.
- `404.php` is the 404 template, used when a post, page, or other content, is not found.

3.3 Static Site Generators

When a user visits a website, expecting the latest content, the web server queries the database layer to get the content, pass the results to the templating engine, that compose the HTML, and let the server serve the dynamically generated page.

Static site generators shift the heavy load from the moment users request the webpage to the moment the webpage actually changes, when the website is updated, generating a structure of purely static HTML files that are ready to be delivered as are to the users.

Static site generators seem to have been becoming more and more popular, but they're not one of those ephemeral novelty things that grow in popularity as quickly as they fall into oblivion shortly after. There's a strong open source community maintaining and pushing forward a wide range of engines with different flavours and features. For over a decade, many different projects (more than 400[139]) have been built with a wide range of programming languages and technologies.

Hosting static websites

Static websites can be hosted anywhere, including hosting services like Netlify[140], Heroku[141], GitHub Pages[142], GitLab Pages[143], Surge[144], Firebase[145], Google Cloud Storage[146], Amazon S3[147], Azure[148], and CloudFront[149] and work well with CDNs.

Static websites run without the need for a database or dependencies on server runtimes like Ruby, Python, or PHP.

---

139 StaticSiteGenerators.net - *The definitive listing of Static Site Generators.* (https://goo.gl/T4Mnyp)
140 *Netlify Docs.* (https://goo.gl/V6EDTM)
141 *Getting Started on Heroku.* (https://goo.gl/cCbw6F)
142 *What is GitHub Pages?* (https://goo.gl/9Zm8c2)
143 *GitLab Pages.* (https://goo.gl/rdAXhV)
144 *Getting started with Surge.* (https://goo.gl/PjkEKn)
145 *Firebase Hosting.* (https://goo.gl/NR7CNG)
146 *Google Cloud Platform Documentation.* (https://goo.gl/QrxkzC)
147 *Amazon S3.* (https://goo.gl/7s6qPs)
148 Andrew Coates (January, 2016). *Publish a static web site using Azure Web Apps.* (https://goo.gl/WzjjAX)
149 *Amazon CloudFront.* (https://goo.gl/JjGDpm)

3.3.1 Content

Essentially, static site generators take the content, typically stored in flat files rather than in databases, pass it to a templating engine, and generate a static version of the website.

The content source is often written in Markdown.

Markdown

Markdown is an easy-to-read/easy-to-write plain text formatting syntax, that can be converted to structurally valid HTML.

Markdown is intended to be as easy-to-read and easy-to-write as is feasible. The overriding design goal for Markdown formatting syntax is to make it as readable as possible.

A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown syntax has been influenced by several existing text-to-HTML filters, such as Textile and reStructuredText, the single biggest source of inspiration for Markdown syntax is the format of plain text email.

Therefore, Markdown syntax is comprised entirely of punctuation characters, which have been carefully chosen so as to look like what they mean. E.g., asterisks around a word actually look like *emphasis*.

Markdown syntax is intended for one purpose: to be used as a format for writing for the web.

Markdown is not a replacement for HTML, or even close to it. Its syntax is very small, corresponding only to a very small subset of HTML tags. The idea is not to create a syntax that makes it easier to insert HTML tags.

The idea for Markdown is to make it easy to read, write, and edit prose. HTML is a publishing format; Markdown is a writing format. Thus, Markdown formatting syntax only addresses issues that can be conveyed in plain text.

For any markup that is not covered by Markdown syntax, you can use HTML itself. There's no need to preface it or delimit it to indicate that you're switching from Markdown to HTML; you just use the tags.

The only restrictions are that block-level HTML elements (e.g. `<div>`, `<table>`, `<pre>`, `<p>`, etc.), must be separated from surrounding content by blank lines, and the start and end tags of the block should not be indented with tabs or spaces. Markdown is smart enough not to add extra (unwanted) `<p>` tags around HTML block-level tags.

Markdown formatting syntax is not processed within block-level HTML tags.
For example, you can't use Markdown-style `*emphasis*` inside a `<div>` block.

Markdown formatting syntax is only processed within span-level HTML tags.
For example, you can use Markdown-style `**strong**` inside a `<em>` span.

For example:

```
# This is a title
## This is a subtitle

The following is a list:
item one
item two
item three

<table>
  <tr>
    <td>This is a table</td>
  </tr>
</table>

This is a regular **strong** paragraph.
```

Shortcodes

Static site generators use Markdown because of its simple content format, but there are times when Markdown falls short. Often, content authors are forced to add raw HTML (e.g., video, custom elements) to Markdown content. But this contradicts the beautiful simplicity of the syntax.

To circumvent these limitations, Hugo created shortcodes.

A shortcode is a simple snippet inside a content file used to render a predefined template. Shortcodes do not work in template files but directly in content file. If you need the type of drop-in functionality that shortcodes provide but in a template, a partial template[150] should be used instead.

In content files, a shortcode can be called by calling:

```
{{% shortcode-name parameters %}}
```

Shortcode parameters are space delimited, and parameters with internal spaces can be quoted. Depending upon how the shortcode is defined, the parameters may be named, positional, or both, although you can't mix parameter types in a single call. The format for named parameters is name="value", similar to HTML attributes.

Some shortcodes use or require closing shortcodes.

For example:

```
{{% highlight javascript %}} javascript code here {{% /highlight %}}
```

[150] *Partial Templates*. (https://goo.gl/tkk7ts)

Frontmatter

Since there is no database, metadata are written with the content, in a dedicated section at the beginning of the file, called frontmatter.

The value for slug is determined by the name of the content file, but it can be overridden.
When defined in the front matter, the slug can take the place of the filename.
For example, the following `content/posts/old-post.md` will be `/posts/new-post/`:

```
---
title: New Post
slug: "new-post"
---

# This is the title
This is the content
```

The content type is determined by the location of the content file, but it can be overridden.
For example, the following content `content/posts/my-post.md`
will use `layouts/new/mylayout.html` to render the content:

```
---
title: My Post
type: new
layout: mylayout
---

# This is the title
This is the content
```

3.3.2 Site structure

In general, the same structure that works to organize the source content is used to organize the rendered site: the organization of the source content will be mirrored in the destination.

For example:

```
content
├── pages
|    ├── _index.md
|    ├── page_1.md
|    ├── page_2.md
|    └── subpages
|         └── subpage_1.md
└── posts
     ├── _index.md
     ├── post_1.md
     └── post_2.md
```

Where:
- `/content/about/_index.md` becomes `/about.html`
- `/content/posts/post-1.md` becomes `/posts/post-1.html`
- `/content/posts/nested/nested-post.md` becomes `/posts/nested/post-2.html`
- `/content/quotes/quote-1.md` becomes `/quotes/quote-1.html`

`_index.md` file allows you to add front matter and content to your list templates[151], such as section templates[152], taxonomy templates[153], taxonomy terms templates.
You can keep one `_index.md` in the content folder for the homepage, and one in each of the content sections, taxonomies, and taxonomy terms.

Single content files in each sections are going to be rendered as single page templates[154].

[151] *Lists of Content in Hugo*. (https://goo.gl/2682Xi)
[152] *Section Page Templates*. (https://goo.gl/LmVSDb)
[153] *Taxonomy Templates*. (https://goo.gl/wr7iEc)
[154] *Single Page Templates*. (https://goo.gl/rpPuis)

Page Bundles

Page Bundles are a way to group page resources[155], such as pages, images, documents etc.
A Page Bundle can be a Leaf Bundle or a Branch Bundle.

A Leaf Bundle is any directory at any hierarchy within the `content` directory that contains an `index.md` file and no children. The hierarchy depth at which a leaf bundle is created does not matter, as long as it is not inside another leaf bundle.

A Branch Bundle is any directory at any hierarchy within the `content` directory, that contains at least an `_index.md` file (e.g. home page, section, taxonomy terms, taxonomy list).
The hierarchy depth at which a branch bundle is created does not matter.

For example:

```
content/
├── branch-bundle-1
|   ├── branch-content-1.md
|   ├── branch-content-2.md
|   ├── image1.jpg
|   ├── image2.png
|   └── _index.md
└── branch-bundle-2
    ├── _index.md
    └── a-leaf-bundle
        └── index.md
```

[155] *Page Resources*. (https://goo.gl/QMuA3E)

3.3.3 Content types

The top level folder, called the root sections, determine the content type.

Each new piece of content you place into a section will automatically inherit the type of the section.
A content type can have a unique set of metadata (i.e., front matter[156]) or customized template and can be created via archetypes[157].

For example, a new file created at `content/posts/new-post.md` will automatically be assigned the type `posts`, alternatively, you can set the content type in a content file's front matter in the field `type`.

To create a new content type, you have to define the templates and archetype unique to the new content type. If you do not specifically declare content types in your front matter or develop specific layouts for content types, the static site generator will assume the content type from the file path and section.

[156] *Front Matter*. (https://goo.gl/7MdRdS)
[157] *Archetypes*. (https://goo.gl/5jYZAF)

Archetypes

Archetypes are templates used when creating new content that contains preconfigured frontmatter and possibly also a content disposition for your website's content types.

Hugo uses the content-section to find the most suitable archetype template in your project. If your project does not contain any archetype files, it will also look in the theme.

For example, for the content in `content/posts/my-first-post.md`, Hugo will use the first archetype file found of these:

- `archetypes/posts.md`
- `archetypes/default.md`
- `themes/{theme-name}/archetypes/posts.md`
- `themes/{theme-name}/archetypes/default.md`

For example, the following archetype stored in `archetypes/new-archetype.md` will create a new `new-archetype` type of content file based on the archetype template.

```
---
title: "{{ replace .Name "-" " " | title }}"
date: {{ .Date }}
draft: true
---

**Insert Lead paragraph here.**

## New posts

{{ range first 10 ( where .Site.RegularPages "Type" "new-type" ) }}
* {{ .Title }}
{{ end }}
```

3.3.4 Taxonomies

Hugo includes support for user-defined groupings of content called taxonomies.
Taxonomies are classifications of logical relationships between content.
A Taxonomy is a categorization that can be used to classify content.
A term in a taxonomy is a key within the taxonomy.
A value in a taxonomy is a piece of content assigned to a term.

When taxonomies are used—and taxonomy templates are provided—Hugo will automatically create both a page listing all the taxonomy's terms and individual pages with lists of content associated with each term.

For example, a website about movies could include the taxonomies, such as Actors, Directors, Studios, Genre. Year, Awards, etc. If, for each of your movies, you specify terms for each of these taxonomies (i.e., in the front matter of each of your movie content files), Hugo would automatically create pages for each each of these taxonomies, with each listing all of the pages that matched that specific taxonomy.

Hugo natively supports taxonomies, for tags and categories.

For example, a taxonomy for categories A, B and C, will create a single page (at `/categories`) that lists all the terms within the categories A, B and C and individual taxonomy list pages for each of the categories that shows a listing of all pages marked as part of that taxonomy within any content file's front matter (at `/categories/A`, `/categories/B`, and `/categories/C`).

3.3.5 Themes

A theme consists of templates and static assets, such as JavaScript and CSS files.

Hugo provides a theming system.
The community-contributed themes are hosted in a centralized GitHub repository[158].

To use a theme, you have to install the theme in the /themes directory.
Hugo applies the decided theme first and then applies anything that is in the local directory. This allows for easier customization while retaining compatibility with the upstream version of the theme.

To customize a theme, you should not edit the theme's files directly, but you should override theme layouts and static assets in your top-level project directories. Hugo permits you to supplement or override any theme template or static file with files in your working directory. This provides the added flexibility of tweaking a theme to meet your needs while staying current with a theme's upstream.

Anytime Hugo looks for a matching template, it will first check the working directory before looking in the theme directory. To modify a template, simply create that template in your local layouts directory. The template lookup order[159] determines which template to use for a given piece of content.

---

[158] *Hugo themes. (*https://goo.gl/mqkgYg)
[159] *Hugo's Lookup Order.* (https://goo.gl/NdXJZr)

Layout templates

Fundamentally, website content is displayed in two different ways, a single piece of content and a list of content items.

With Hugo, a theme layout starts with the defaults. As additional layouts are defined, they are used for the content type or section they apply to. This keeps layouts simple, but permits a large amount of flexibility.

The default single file layout is located at `layouts/_default/single.html`.
The default list file layout is located at `layouts/_default/list.html`.

Partial templates

Theme creators should liberally use partial templates throughout their theme files. Not only is a good DRY (Don't Repeat Yourself) practice to include shared code, but partials are a special template type that enables the themes end user to be able to overwrite just a small piece of a file or inject code into the theme from their local `/layouts`. These partial templates are perfect for easy injection into the theme with minimal maintenance to ensure future compatibility.

Static files

Everything in the static directory will be copied directly into the final site when rendered. No structure is provided here to enable complete freedom. It is common to organize the static content into `/css`, `/js` and `/img` folders.

3.4 Decoupled Web Content Management Systems

3.4.1 Decoupled Web CMS

Choosing a Web CMS means accepting not only the language it's written in, but also its editing and administration tools, its database, its templating system, etc.
Decoupled Web CMSs aim to improve this situation[160].

There is a trend for software architecture that tends to decouple the parts of a system.
A Decoupled CMS is essentially a regular full stack of content management, delivery, and presentation solution but allows for content stored within it to be leveraged by other systems.

Tightly-coupled vs loosely-coupled systems

In any system, from a urban infrastructure to a computer program, the designer of the system can choose the degree to which the pieces of the system depend on one another[161].

In a tightly-coupled system, every piece depends each other; each part of the system can make assumptions about the other parts; if one piece fails, it could take the whole system with it.
In a loosely-coupled system, all the pieces are independent; each part has little to no knowledge of the other pieces; individual parts of the system can be swapped out with a minimum of knock-on effects.

Tightly-coupled systems can be designed quite quickly, but at a price: they lack resilience.
Loosely-coupled system can take more work to be designed, but with a payoff: the overall result is more resilient to failure.

Decoupled software architectures come with a wide range of advantages and promises. They are better at fulfilling the requirements for flexibility and for agility as well as the ever growing need for scalability. This trend applies to all sort of software, especially for Web architectures and systems.

---

[160] D. Buchmann. (March 2015). *The benefits of decoupling your CMS.* (https://goo.gl/gFx59L)
[161] J. Keith. (2015). *Resilient web design.* (https://goo.gl/MvnUFv)

3.4.2 Headless CMS

A Headless CMS is a Decoupled CMS where the content management repository system is independent of the content delivery and presentation tools.

While traditional CMSs have to be hosted and built together with the website every time it's served, a Headless CMS doesn't care where it's serving its content to, since it's no longer attached to the frontend[162].

In a Headless CMS, the content for the site is accessible via a web-service API, usually in a RESTful manner and in a mashup-friendly format such as JSON; the end-user experience is delivered by a Javascript application rendering the output of this API into HTML, frequently making use of a modern application framework.

Typically, the content is stored in a NoSQL database, and handled via HTTP API.

[162] T. Schadler and M. Grannan. (Mach 2016). *The Rise Of The Headless Content Management System*. (https://goo.gl/KLrk1q)

3.5 Static Progressive Web App Generators

Static progressive Web app generators offers the advantages of a static website generator and the versatility of an API driven Headless CMS plus a front-end Web Framework.

3.5.1 Gatsby.js

Gatsby is static progressive Web App generator written in JavaScript.

Gatsby follows the PRPL architectural pattern. It renders a static HTML version of the initial route, then it loads the code bundle for the page, then it starts pre-caching resources for pages linked to from the initial route. When a user clicks on a link, Gatsby creates the new page on demand on the client.

PRPL Pattern

PRPL is a web site architecture developed by Google for building websites and apps that work exceptionally well on smartphones and other devices with unreliable network connections[163].

PRPL stands for "Push, Render, Pre-cache, Lazy-load" that mean:

- Push critical resources for the initial URL route using <link preload> and http/2.
- Render initial route.
- Pre-cache remaining routes.
- Lazy-load and create remaining routes on demand.

[163]Addy Osmani (*February, 2018). The PRPL Pattern.* (https://goo.gl/Tnk9Fe)

Directory structure

A basic directory structure of a project might look like this:

```
.
├── gatsby-config.js
├── package.json
└── src
    ├── html.jsx
    ├── pages
    │   ├── index.jsx
    │   └── posts
    │       ├── post_1
    │       │   └── index.md
    │       ├── post_2
    │       ... └── index.md
    ├── templates
    │   └── post.jsx
    └── layouts
        └── index.jsx
```

Where:

- `gatsby-config.js` exports the configuration object
- `package.json` is the configuration for the Node.js package dependencies
- `src` directory contains the source for the app

Where, in `src`:

- `html.jsx` is used to define the HTML document that host the page components
- `pages` directory contains the page components
- `templates` directory contains the page template components
- `layouts` directory container the page layout components

3.5.2 Components

Everything in Gatsby is built using components, i.e. React Components.
Gatsby automatically tracks dependencies between different objects, e.g. a component page can depend on certain nodes, enabling hot reloading, caching, incremental rebuilds, etc.
Gatsby support .js and .jsx component, but can also support other compile-to-js languages through plugins.

Page Component

A page is a site page with a pathname, a template Page component.
A Page Component is a React component that renders a page.
It can optionally specify a layout component and a graphql query.
Page components are stored in `src/pages` directory.
Every .js or .jsx file in the `src/pages` directory must resolve to a react component, or a string.

For example, in `src/pages/index.jsx`:

```
export default function IndexPage {
   return (
    <div className="container">
      <h1>Welcome!</h1>
      <a href="/about">Read more about me.</a>
    </div>
  )
}
```

For example, in `src/pages/about.jsx`:

```
export default function AboutPage {
   return (
    <div className="container">
      <h1>About me</h1>
    </div>
  )
}
```

Page components become pages automatically with paths based on their file name.
For example:

- `src/pages/index.jsx` is mapped to `http://example.org/`
- `src/pages/about.jsx` is mapped to `http://example.org/about/`

Page template components

All pages are React components, but very often these components are just wrappers around data from files or other sources. You can programmatically create pages using page template components.

Page template components are stored in `src/templates` directory:

For example:

```
export default function PostTemplate (queryResult) {
  const post = queryResult.data.markdownRemark
  return (
    <div>
      <h1>{post.frontmatter.title}</h1>
      <div dangerouslySetInnerHTML={{ __html: post.html }} />
    </div>
  )
}
```

```
export const pageQuery = graphql`
  query BlogPostBySlug($slug: String!) {
    markdownRemark(fields: { slug: { eq: $slug } }) {
      html
      frontmatter {
        title
      }
    }
  }
`
```

Where:

- `pageQuery` queries GraphQL for markdown data
- `PostTemplate` renders the page using the query result

Layout components

Layout components wraps page components. You can use it for portions of pages that are shared across pages, like headers and footers.
Layout components are stored in d `src/layouts` directory.

For example:

```
export default class Template extends React.Component {
  render() {
    return (
      <header />
      <div>{this.props.children</div>
      <footer />
    )
  }
}
```

HTML component

The component stored in `src/html.jsx` is responsible for the final HTML page where the page component is injected.
In this file you can modify the `<head>` metadata, general structure of the document and add external links, such as JavaScript, CSS, etc.
Typically, the default provided `html.js` file is suffice. If you need more control over server rendering, then it's valuable to have an html.js.

For example:

```
export default class HTML extends React.Component {
  render() {
    return (
      <html lang="en">
        <head>
          <meta charSet="utf-8" />
          <meta name="viewport" content="width=device-width,initial-scale=1.0" />
          {this.props.headComponents}
        </head>
        <body>
          <div id="___gatsby"
            dangerouslySetInnerHTML={{ __html: this.props.body }} />
          {this.props.postBodyComponents}
        </body>
      </html>
    )
  }
}
```

3.5.3 Data Management

Data Nodes

All data that's added to Gatsby is modeled using nodes.

The "Node" is the center of Gatsby's data system.

A Node is simply a data object.
Node fields can added to a node by plugins that the node does not control.
Node links, which are connection between nodes, gets converted to GraphQL relationships.
Node links can be created in a variety of ways as well as automatically inferred.
Parent/child links from nodes and their transformed derivative nodes are first class links.

Nodes are created by "source" plugins.

Existing nodes can be transformed to new type of nodes by "transformer" plugins.
Nodes created by transformer plugins are set as "children" of their "parent" nodes.
For example, the Remark (Markdown library) transformer plugin looks for new nodes that are created with a mediaType of `text/markdown` and then transforms these nodes into `MarkdownRemark` nodes which have an `html` field.
For another example, the YAML transformer plugin[164] looks for new nodes with a media type of `text/yaml` (e.g. a `.yaml` file) and creates new YAML child node(s) by parsing the YAML source into JSON objects.

[164] *Gatsby-transformer-yaml Plugin. (*https://goo.gl/GwUGFC*)*

The basic node data structure is as follows:

```
{
  id: String,
  children: Array[String],
  parent: String,
  fields: Object,
  internal: {
    contentDigest: String,
    mediaType: String,
    type: String,
    owner: String,
    fieldOwners: Object,
    content: String
  }
}
```

Where:

- `fields` is reserved for plugins who wish to extend other nodes.
- `mediaType` is optional field that indicates this node has data to be further processed.
- `type` is a globally unique node type chosen by the plugin owner.
- owner is the plugin which created this node
- `fieldOwners` stores which plugins created which fields.
- `content` is an optional field exposing the raw content that can be further processed.

Querying Data Nodes with GraphQL

There are many options for loading data into React components. One of the most popular and powerful of these is a technology called GraphQL.

GraphQL is a query language (the QL part of its name). It was invented at Facebook to help product engineers pull needed data into React components.

GraphQL is usually run on a server to respond live to requests for data from clients. You define a schema for your GraphQL server and then your GraphQL resolvers retrieve data from databases and/or other APIs.

Gatsby uses GraphQL at build-time and not for live sites. This is unique, and it means you don't need to run additional services (e.g. a database and node.js service) to use GraphQL for production websites. This makes Gatsby sites serverless.

Gatsby uses GraphQL to enable page and layout components to declare what data they and their sub-components need. Then, Gatsby makes that data available in the browser when needed by your components. Using a special syntax, you describe the data you want in your component and then that data is given to you.

GraphQL lets you ask for the exact data you need.

For example, the following GraphQL query return the following JSON:

```
{
  site {
    siteMetadata {
      title
    }
  }
}
```

```
{
  "site": {
    "siteMetadata": {
      "title": "Hello GraphQL!"
    }
  }
}
```

For example, a basic page component with a GraphQL query might look like this:

```
export default function AboutPage ({ data }) {
  return (
    <div>
      <h1>About {data.site.metadata.title}</h1>
      <p>{data.site.metadata.description}</p>
    </div>
  )
}
```

```
export const queryPage = graphql`
  query AboutQuery {
    site {
      siteMetadata {
        title,
        description
      }
    }
  }
`
```

The result of the query is automatically inserted into the component on the data prop. GraphQL and Gatsby let you ask for data and then immediately start using it.

Note that queries are only executed from Page or Layout components.

Gatsby’s GraphQL schema

Most usages of GraphQL involve manually creating a GraphQL schema.
With Gatsby, you use (source) plugins which fetch data from different sources.
Gatsby use that data to automatically infer a GraphQL schema.

For example, if you give Gatsby the following data, Gatsby will create the following schema:

```
{
  "title": "A long long time ago"
}
```

```
title: String
```

This makes it easy to pull data from anywhere and immediately start writing GraphQL queries against your data.

This can cause confusion as some data sources allow you to define a schema even when there’s not any data added for parts or all of the schema. If parts of the data haven’t been added, then those parts of the schema might not be recreated in Gatsby.

3.5.4 Plugins

The Bootstrap processes

Gatsby has multiple processes. The most prominent is the "Bootstrap" process, that has several sub-processes. These processes run both once during the initial bootstrap but also stay alive during development to continue to respond to changes. This is what drives hot reloading that all Gatsby data is "alive" and reacts to changes in the environment.

The Bootstrap process consists of the following processes:

1. load site config
2. load plugins
3. load source nodes
4. load transform nodes
5. create graphql schema
6. create pages
7. compile component graphql queries
8. run graphql queries
9. react to changes

During these processes there are various extension points where plugins can intervene.
During bootstrap, plugins can respond at various stages to APIs like `onCreatePages`, `onSourceNodes`. All major processes have a `onPre-` and `onPost-` extension points, e.g. `onPreBootstrap` and `onPostBootstrap`, or `onPreBuild` or `onPostBuild`.

At each extension point, Gatsby identifies the plugins which implement the API and calls them in serial following their order in the site's gatsby-config.js.

Source plugins

Source plugins create new nodes.

For example, `gatsby-source-filesystem`[165] plugin turns files on disk into `File` nodes.

Transformer plugins

Transformer plugins create new types of nodes by transforming source nodes.

A transform plugin looks at the media type of every new node, that is created by the source plugins, to decide if it can transform the new node or not.

Transformer plugins are decoupled from source plugins.
For example, a markdown transformer plugin can transform markdown from any source —without any other configuration— provided the created node supports markdown in some of its data fields.

[165] *Gatsby-source-filesystem Plugin*. (https://goo.gl/EoVhkp)

3.5.5 Building a Blog site

Building a Blog site in Gatsby includes the following tasks:

- Read (from the filesystem) and transform markdown files.
- Create the Page Template Components for the Post and the Tag nodes.
- Create the static pages

The Page Template Components to create are:

- `PostsTemplate` (for `/posts` page)
- `PostTemplate` (for `/posts/{post}` pages)
- `TagsTemplate` (for `/tags` page)
- `TagTemplate` (for `/tags/{tag}` pages)

The static pages are created at build time, in the `gatsby-node.js` file.

Read and transform markdown files

Using `gatsby-source-filesystem` plugin, you can read files from the file-system.
Using the `gatsby-transformer-remark` plugin, you can transform the markdown content to HTML and the YAML frontmatter to JSON data.

The Gatsby configuration object will result:

```
{
  plugins: [
    {
      resolve: `gatsby-source-filesystem`,
      options: {
        path: `${__dirname}/path/to/markdown/files`,
        name: "markdown-pages",
      },
    },
    `gatsby-transformer-remark`
  ]
}
```

Posts page

In `src/pages/index-page.jsx`:

```
export default function IndexPage (queryResult) {
  const data = queryResult.data
  const allMarkdownRemark = data.allMarkdownRemark
  const edges = allMarkdownRemark.edges
  const posts = edges.filter(({node}) => (!!node.frontmatter.date))

  return (
    <div>
      {posts.map(({node}) => (
        <Link key={node.id} to={node.post.frontmatter.path} />
          {node.post.frontmatter.title} ({node.post.frontmatter.date})
        </Link>
      ))}
    </div>
  )
}
```

```
export const pageQuery = graphql`
  query IndexQuery {
    allMarkdownRemark(sort: { order: DESC, fields: [frontmatter___date] }) {
      edges {
        node {
          id
          excerpt(pruneLength: 250)
          frontmatter {
            date(formatString: "MMMM DD, YYYY")
            path
            title
          }
        }
      }
    }
  }
`
```

Post page

In the `/src/templates/postTemplate.jsx`:

```
export default function PostTemplate (queryResult) {
  const data = queryResult.data
  const content = data.markdownRemark
  const frontmatter = content.frontmatter
  const html = content.html;
  return (
    <div className="blog-post-container">
      <div className="blog-post">
        <h1>{frontmatter.title}</h1>
        <h2>{frontmatter.date}</h2>
        <div className="blog-post-content"
          dangerouslySetInnerHTML={{ __html: html }}></div>
      </div>
    </div>
  )
}
```

```
export const pageQuery = graphql`
  query PostByPath($path: String!) {
    markdownRemark(frontmatter: { path: { eq: $path } }) {
      html
      frontmatter {
        date(formatString: "MMMM DD, YYYY")
        path
        title
      }
    }
  }
`
```

Tags page

In `src/pages/tags-page.jsx`:

```
export default function TagsPage (queryResult) {
  const data = queryResult.data
  const group = data.allMarkdownRemark.group

  return (
    <div>
      <h1>Tags</h1>
      <ul>
        {group.map(tag => (
          <li key={tag.fieldValue}>
            <Link to={`/tags/${kebabCase(tag.fieldValue)}/`}>
              {tag.fieldValue} ({tag.totalCount})
            </Link>
          </li>
        ))}
      </ul>
    </div>
  )
}
```

```
export const pageQuery = graphql`
  query TagsQuery {
    allMarkdownRemark(
      limit: 1000
      filter: { frontmatter: { published: { ne: false } } }
    ) {
      group(field: frontmatter___tags) {
        fieldValue
        totalCount
      }
    }
  }
`;
```

Tag page

In `src/templates/tag-page.js`:

```
export default function TagPage (queryResult) {
  const pathContext = queryResult.pathContext
  const tag = pathContext.tag
  const data = queryResult.data
  const edges = data.allMarkdownRemark

  return (
    <div>

      <ul>
        {edges.map(({ node }) => (
            <li key={node.id}>
              <Link to={node.frontmatter.path}>
                {node.frontmatter.title}
              </Link>
            </li>
          )
        )}
      </ul>
      <Link to="/tags">All tags</Link>
    </div>
  )
}
```

```
export const pageQuery = graphql`
  query TagPage($tag: String) {
    allMarkdownRemark(
      limit: 2000
      sort: { fields: [frontmatter___date], order: DESC }
      filter: { frontmatter: { tags: { in: [$tag] } } }
    ) {
      totalCount
      edges {
        node {
          frontmatter {
            title
            path
          }
        }
      }
    }
  }
`
```

Create pages

```
export default async function createPages(app) {
  const query = `
    {
      allMarkdownRemark(
        sort: { order: DESC, fields: [frontmatter___date] }
        limit: 1000
      ) {
        edges {
          node {
            frontmatter {
              path
              tags
            }
          }
        }
      }
    }`

  const queryResult = await app.graphql(query)
  if (queryResult.errors) {
    return queryResult.errors
  }

  const postPageTemplate = path.resolve("src/templates/postPage.js")
  _createPostPages(app, queryResult, postPageTemplate)

  const tagPageTemplate = path.resolve("src/templates/tagPage.js")
  _createTagPages(app, queryResult, tagPageTemplate)
}
```

```
function _createPostPages (app, queryResult, component) {
  const posts = queryResult.data.allMarkdownRemark.edges
  posts.forEach(({ node }) => {
    app.boundActionCreators.createPage({
      path: node.frontmatter.path,
      component: blogPostTemplate
    })
  })
}
```

```
function _createTagPages (app, queryResult, component) {
  const posts = queryResult.data.allMarkdownRemark.edges
  const tags = []

  posts.forEach(({ node }) => {
    node.frontmatter.tags.forEach(tag => {
      if (!tags.includes(tag)) {
        tags.push(tag)
    })
  })

  tags.forEach(tag => {
    app.boundActionCreators.createPage({
      path: `/tags/${tag}/`,
      component: tagTemplate,
      context: { tag }
    })
  })
}
```

CHAPTER 4
FRONT-END WEB FRAMEWORKS

When Tim Berners Lee invented the web he was looking for a system to publish scientific documents remotely accessible, easy to code and easy to use for a non-technical person. For this reason, the Web was conceived as a document based system with pages and links.

Over the years, there has been a strong effort to adapt the web paradigm of pages and links to more sophisticated application development, to enable the Web as a platform for web-based applications.

As much as the web platform has changed, so have the needs of tools we use to build the applications.

Over the past couple of years, we have seen the rise of the most mature set of frameworks. A virtual cornucopia of options. By and large, these all focus on creating extremely rich UI/UX experiences that work across the gamut of devices from desktop to mobile. Some frameworks provide highly structured ways to build applications and some focus on solving just a single problem, expecting to be incorporated into other frameworks to build a whole application.

4.1 Single Page Applications

A single-page application (SPA) is a website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This approach avoids interruption of the user experience between successive pages, making the website behave more like a desktop application.

In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions.

The single page does not reload at any point in the process, nor does control transfer to another page, although the location hash or the HTML5 History API[166] can be used to provide the perception and navigability of separate logical pages in the application.

Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

In general, to ease the development of an application, a frameworks and programming libraries are often used. Similarly, to ease the development of a Single Page Application, a Front-end Web Framework is often used.

[166] W3C. *The History API Spec.* (https://goo.gl/BfQEks)

Frameworks vs. Libraries

A library is a collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which the behavior is invoked[167].
In Web development, a library, i.e. a JavaScript library, is a collection of JavaScript functions, that define specific operations in a domain specific area.
For example, mathematics libraries let you use complex mathematical functions without redo the implementation of how an algorithm works.

A framework is an abstraction, in which software, providing generic functionality but suggesting a particular workflow, can be selectively changed by additional user-written code, to provide application-specific software.
In Web development, a framework, i.e. a JavaScript framework, is a collection of interdependent classes that define a skeleton to build structured Web applications.
For example, Web UI Frameworks let you build Web UI, focusing on what you want to design, hiding the complexity of data management, event handling, scoped styling, etc.

[167] Hubert Narożny (October 2016). *Framework vs Library - differences in web development*. (https://goo.gl/wMRbPm)

4.1.1 MVC Front-end Web Frameworks

Traditionally used for desktop graphical user interfaces (GUIs), MCV architecture has become popular for designing Web applications.

MVC (which stands for Model–View–Controller) is commonly used for developing software that divides an application into three interconnected parts: models, views and controllers. This is done to separate internal representations of information, the model, from the ways information is presented to, the views, and accepted from the user, the controller.

Several Web frameworks have been created in the MVC architecture. These frameworks vary in their implementation, mainly in the way the MVC responsibilities are divided.

The first front-end Web framework that implemented the MVC pattern was Backbone.js. Since Backbone.js, various frameworks have been created: actually, TodoMVC[168] — a project which offers the same Todo application implemented using MV* concepts in most of the popular JavaScript MV* frameworks of today — counts 64 front-end Web frameworks.

[168] TodoMVC. *Helping you select an MV* framework*. (https://goo.gl/gJBLw2)

MVC pattern

As with other software patterns, MVC expresses the "core of the solution" to a problem while allowing it to be adapted for each system[169].

The model is the central component of the pattern:

- it expresses the application's behavior in terms of the problem domain;
- it directly manages the data, logic and rules of the application;
- it is independent of the user interface;
- it receives user input from the controller.

The view can be any output representation of information (the model):

- it receives the model and use the model to render a part of the app.

The controller, accepts input and converts it to commands for the model or the view:

- it receives user input and passes them to the model.

The advantages of MVC are:

- *Simultaneous development* – different developers can work simultaneously on the different components, i.e. models, views and controllers.
- *High cohesion* – related actions on a controller are logically grouping together. The views for a specific model are also grouped together.
- *Low coupling* – The very nature of the MVC framework is such that there is low coupling among models, views or controllers.
- *Ease of modification* – Because of the separation of responsibilities, future development or modification is easier.

The disadvantages of MVC could be:

- *Code navigability* – the code can be complex since it introduces new layers of abstraction and requires to adapt to the decomposition criteria of MVC.
- *Multi-artifact consistency* – decomposing a feature into three artifacts could cause scattering, since it requires to maintain the consistency of multiple parts at once.

[169] Steve Burbeck (1992). *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).* (https://goo.gl/bRKNBv)

Observer pattern

One of the motivations of using the MVC pattern is to make the model independent from of the views. However, if the model had to notify the views of changes, you would reintroduce the dependency you were looking to avoid. This is where the Observer pattern come in.

The Observer pattern provides a mechanism to alert other objects of state changes without introducing dependencies on them.

The individual views implement the Observer interface and register with the model.

The model tracks the list of all observers that subscribe to changes.
The model never requires specific information about any views.
When the model changes (updates its data), it iterates through all registered observers and notifies them of the change.

This approach is often called “publish-subscribe”.

For example, a simple implementation of the view:

```
class View () {

  constructor (model) {
    model.register(this)
  }

  render (data) {
    //render view
  }

}
```

For example, a simple implementation of the model:

```
class Model () {

  constructor () {
    this.listeners = new Set()
    this.data = {}
  }

  register (listener) {
    this.listeners.add(listener)
  }

  notify () {
    for (let listener of this.listeners) {
      listener.render(this.data)
    }
  }

  update (data) {
    //update view
    this.notify()
  }
}
```

With this generic way of communicating between the subject and observers, collaborations can be built dynamically instead of statically. Due to the separation of notification logic and synchronization logic, new observers can be added without modifying the notification logic, and notification logic can also be changed without affecting the synchronization logic in observers. The code is now much more separate, and thus easier to maintain and reuse.

4.1.2 Templating System

To simplify the process of rendering model data into display-ready markup, a templating system is commonly used.

A templating system is composed by:

- a template engine - the primary processing component of the system.
- a template resource - specified according to a template language.
- a content resource - any of various kinds of input data.

The template engine combine the template resource and the content resources to produce the HTML code.

Using a templating system simplifies the dynamic rendering of HTML, especially as the size and complexity of the rendering increases.

For example, the following template resource, combined with the following content resource, will produce the following HTML code.

```
<h1>{{ data.message }}</h1>
<h2>I’m <em>{{ data.author }}</em></h2>
```

```
{
  "message": "Hello Wonderland!"
  "author": "Alice"
}
```

```
<h1>Hello Wonderland!</h1>
<h2>I’m <em>Alice</em></h2>
```

4.1.3 Dynamic rendering

There are two techniques to dynamically update the template:

- Data binding
- Virtual DOM

Data binding

Data binding is the process that establishes a connection between the view and the model.

If the binding has the correct settings and the data provides the proper notifications, then, when the data changes its value, the elements that are bound to the data reflect changes automatically.

Actually, the most used UI Framework driven by data-binding is Angular[170] by Google, and Vue.js[171] by Evan You.

Virtual DOM

Actually, the most (and the first) used UI Framework powered by Virtual DOM is React.js[172] by Facebook.

---

[170] Angular.io. *One Framework. Mobile & Desktop*. (https://goo.gl/pYMtU6)
[171] Vue.js. *The Progressive JavaScript Frameworks*. (https://goo.gl/L33cPi)
[172] React.js. *A JavaScript library for building user interface*. (https://goo.gl/vAAEUb)

4.2 Angular

Angular is a framework for building Web applications in HTML and TypeScript[173]. TypeScript is a typed superset of JavaScript, developed by Microsoft, that compiles to plain JavaScript output[174]. Angular is itself written in TypeScript.

4.2.1 Modules

Angular apps are modular and Angular has its own modularity system called NgModules.

An NgModule is a container for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. It can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—in order to minimize the amount of code that needs to be loaded at startup.

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule and resides in a file named app.module.ts. Every Angular app is launched by bootstrapping the root NgModule.

While a small application might have only one NgModule, most apps have many more feature modules. The root NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.

NgModules provide a compilation context for their components.

[173] Angular. *Architecture overview*. (https://goo.gl/JGphQ4)
[174] Microsoft. *TypeScript*. (https://goo.gl/tTwqqQ)

4.2.2 Components

Every Angular app has at least one component, the root component, that connects a component hierarchy with the page DOM. Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

For example:

```
@Component({
  selector: 'my-component,
  templateUrl: './my-component.component.html'
})
export class MyComponent implements OnInit {
  ngOnInit() {
    /* component is initialized */
  }
}
```

Where:

- `@Component` decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.
- `selector` is a CSS selector that tells Angular to create and insert an instance of the component wherever it finds the corresponding tag in template HTML.
- `templateUrl` is the module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
- `OnInit` indicates the class implements the `ngOnInit` method, that is called when the component is initialized.

Decorators are functions that modify JavaScript classes[175]. Angular defines a number of such decorators that attach specific kinds of metadata to classes, so that it knows what those classes mean and how they should work.

[175] Addy Osmani (July 2015). *Exploring EcmaScript Decorators*. (https://goo.gl/fvCTbY)

4.2.3 Templates

You define a component's view with its companion template. A template is a form of HTML that tells Angular how to render the component.

Views are typically arranged hierarchically, allowing you to modify or show and hide entire UI sections or pages as a unit. The template immediately associated with a component defines that component's host view. The component can also define a view hierarchy, which contains embedded views, hosted by other components.

A view hierarchy can include views from components in the same NgModule, but it also can (and often does) include views from components that are defined in different NgModules.

Template syntax

A template looks like regular HTML, except that it also contains Angular template syntax[176], which alters the HTML based on your app's logic and the state of app and DOM data. Your template can use data binding to coordinate the app and DOM data, pipes to transform data before it is displayed, and directives to apply app logic to what gets displayed.

---

[176] Angular.io. *Template Syntax*. (https://goo.gl/JEXAKL)

4.2.4 Data binding

Angular supports four forms of data binding markup.

Text interpolation

For example:

```
<div id="el">Hello {{name}}!</div>
```

Property binding

For example, the following are equivalent:

```
<a id="el" [title]="message">link</a>
```

```
<a id="el" bind-title="message">link</a>
```

Event handling

For example, the following are equivalent:

```
<a id="el" (click)="onClick()">link</a>
```

```
<a id="el" on-click="onClick()">link</a>
```

Two-way binding

For example, the following are equivalent:

```
<div id="app">
  <input id="input" [(ngModel)]="name">
  <div id="el">Input: {{name}}</div>
</div>
```

```
<div id="app">
  <input id="input" bindon-ngModel="name">
  <div id="el">Input: {{name}}</div>
</div>
```

4.3 Vue.js

Vue (pronounced /vjuː/, like view) is a framework for building Web user interfaces.
The core library is focused on the view layer only, and can be integrated with other libraries.

4.3.1 Data binding

Vue allows you to declaratively render data to the DOM using a special template syntax.

Binding inner text

To dynamically insert text in an element, you use the double-curly-brace syntax.

For example:

```
<div id="element">Hello {{ name }}!</div>
```

```
const el = '#element'
const data = {
  name: 'Vue'
}
const app = new Vue({ el, data })
```

Binding attributes

To bind an element attribute, you use the `v-bind:`*attribute* syntax.

For example:

```
<a id="element" v-bind:title="message">link</a>
```

```
const el = '#element'
const data = { message: "This is the title" }
const app = new Vue({ el, data })
```

Conditional rendering

To toggle an element, you use the `v-if` binding.

For example:

```
<div id="element" v-if="seen">Now you see me</div>
```

```
const el = '#element'
const data = { seen: true }
const app = new Vue({ el, data })
```

Repeat rendering

To repeat an element, you use the `v-if` binding.

For example:

```
<div id="app">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
```

```
const el = '#app'
const data = {
  todos: [
    { text: 'Learn HTML' },
    { text: 'Learn CSS' },
    { text: 'Learn JavaScript' }
  ]
}
const app = new Vue({ el, data })
```

4.4 React

React is a library for building composable user interfaces. It encourages the creation of reusable UI components which present data that changes over time.

React isn't an MVC framework.
React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display. Then, instead of separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called "components" that contain both.

4.4.1 Elements

Elements are the smallest building blocks of React apps.
The smallest React example looks like this:

```
const element = <h1>Hello, world!</h1>
ReactDOM.render(element, document.root)
```

It displays a heading element saying "Hello, world!" on the page.
React apps are usually written in JSX.

React elements are immutable[177]. Once you create an element, you can not change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

React only updates what is necessary.
React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

For example:

```
function Clock () {
  return (
    <div>It is <pre>{new Date().toLocaleTimeString()}</pre></div>
  )
}

function tick () {
  ReactDOM.render(Clock, document.root)
}

setInterval(tick, 1000)
```

In the example, `render()` is called every second from the `setInterval()` callback `tick()`: even though we create an element describing the whole UI tree on every tick, only the text node whose contents has changed gets updated by React DOM.

[177] *Immutable object*. (https://goo.gl/xee6Wd)

4.4.2 JSX

JSX is a syntax extension to JavaScript, that produces React "elements".
React doesn't require using JSX, but when working with UI inside the JavaScript code it become visually necessary[178].

JSX is compiled down to `React.createElement()` calls.

For example:

```
const element = <h1 className="greeting">Hello!</h1>
const compiled = React.createElement('h1', { className: 'greeting' }, 'Hello!')
```

`React.createElement()` essentially creates an object that describe the element.

For example:

```
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
}
```

These objects are called "React elements". You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

Because after compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects, you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions.

---

[178] *React Without JSX*. (https://goo.gl/t7LoUk)

Updating attributes and properties

You can use curly braces to embed a JavaScript expression in an attribute.

For example:

```
const element = <img src={url}></img>
```

Conditional rendering

Conditional rendering in React works the same way conditions work in JavaScript.
You can use JavaScript operators `if`, or the conditional operator, to create elements.

For example:

```
function Greeting (name) {
  if (name) {
    return <h1>Hello, {name}!</h1>
  }
  return <h1>Hello, Stranger.</h1>
}
```

Repeat rendering

Repeat rendering in React works the same way conditions work in JavaScript.
You can use JavaScript Array methods, such as `map`, `filter`, etc., to create elements.

For example:

```
function NumberList (numbers) {
  return (
    <ul>
      {numbers.map(number => (
        <li>{number}</li>
      ))}
    </ul>
  )
}
```

Handling events

Handling events with React elements is very similar to handling events on DOM elements, except some syntactic differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

When using React you should generally not need to call addEventListener to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered. When you define a component using an ES6 class, the event handler should be a method on the class.

For example:

```
class Button extends React.Component {

  handleClick (event) {
    console.log('Touchè')
  }

  render () {
    return (
      <button onClick={this.handleClick.bind(this)}>Click me</button>
    )
  }
}

ReactDOM.render(<Button />, window.root)
```

4.4.3 Components

Defining Components

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

You can also use an JavaScript class[179] to define a component.
For example, the followings produce the same element:

```
function Welcome (props) {
  return <h1>Hello, {props.name}</h1>
}
```

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

```
const element = <Welcome name="Alice" />
```

When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object, called "props".

[179] *Classes*. (https://goo.gl/2jRyMC)

Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.
For example:

```
function Welcome (props) {
  return <h1>Hello, {props.name}</h1>
}

function App () {
  return (
    <div>
      <Welcome name="Alice" />
      <Welcome name="Bob" />
    </div>
  )
}
```

Typically, new React apps have a single App component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like Button and gradually work your way to the top of the view hierarchy.

Component props

Components props are read-only: whether you declare a component as a function or a class[180], it must never modify its own props.
React is pretty flexible but it has a single strict rule: "all React components must act like pure functions with respect to their props".

Of course, application UIs are dynamic and change over time.
The state of a component is represented by another object, called state.

Component state

State allows components to change their output over time in response to user actions, network responses, and anything else.

To change the state, you must call the method `setState()`.
This method lets React to know if the state has changed, and to call the `render()` method.

State updates may be asynchronous.
For performance reasons, React may batch multiple `setState()` calls into a single update.
Because components props and state may be updated asynchronously, you should not rely on their values for calculating the next state.

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
})
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}))
```

State updates are merged.
When you call `setState()`, React merges the object you provide into the current state.
For example, the state may contain several independent variables (e.g. `a` and `b`), but you can update them independently with separate `setState()` calls. The merging is shallow, so `this.setState({ a })` leaves `this.state.b` intact, but completely replaces `this.state.a`.

[180] *Functional and Class Components.* (https://goo.gl/w2otnd)

4.4.4 The data flow

The data flows down.

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class. This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.
A component may choose to pass its state down as props to its child components.

For example:

```
class FormattedDate extends React.Component {
  render () {
    return (<h2>It is {props.date.toLocaleTimeString()}.</h2>)
  }
}

class Clock extends React.Component {
  render () {
    return (<FormattedDate date={this.state.date} />)
  }
}
```

In the example, the FormattedDate component would receive the date in its props and wouldn't know whether it came from the Clock's state, from the Clock's props, or was typed by hand.

The data flows, "top-down" or "unidirectional". Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

4.4.5 The Virtual DOM

When a component is first initialized, the `render()` method is called, generating a lightweight representation of the view. From that representation, a string of markup is produced, and injected into the document. When data changes, the render method is called again.

In order to perform updates as efficiently as possible, the library diff the return value from the previous call to render with the new one, and generate a minimal set of changes to be applied to the DOM. The data returned from render is neither a string nor a DOM node —it's a lightweight description of what the DOM should look like, called the Virtual DOM.

This process is called reconciliation.

Reconciliation

React provides a declarative API so that you don't have to worry about exactly what changes on every update. This makes writing applications a lot easier, but it might not be obvious how this is implemented within React.

When you use React, at a single point in time you can think of the render() function as creating a tree of React elements. On the next state or props update, that render() function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the state of the art algorithms have a complexity in the order of O(n^3) where n is the number of elements in the tree[181]. That is: displaying 1000 elements would require in the order of one billion comparisons.

Instead, React implements a heuristic O(n) algorithm based on two assumptions, that, in practice, are valid for almost all practical use cases:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a `key` prop.

---

[181] Philip Bille. *A Survey on Tree Edit Distance and Related Problems.* (https://goo.gl/D7wBix)

4.5 Redux

As the requirements for JavaScript single-page applications have become increasingly complicated, the application must manage more state than ever before. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on.

Managing this ever-changing state is hard. If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update. At some point, you no longer understand what happens in your app as you have lost control over the when, why, and how of its state. When a system is opaque and non-deterministic, it's hard to reproduce bugs or add new features.

This complexity is difficult to handle as we're mixing two concepts that are very hard for the human mind to reason about: mutation and asynchronicity.

Libraries like React attempt to solve this problem in the view layer by removing both asynchrony and direct DOM manipulation. However, managing the state of your data is left up to you. This is where Redux enters.

Redux is a predictable state container for JavaScript apps. Where predictables means you can alway know how and when states updates happen.

Following in the steps of Flux[182], CQRS[183], and Event Sourcing[184], Redux attempts to make state mutations predictable by imposing certain restrictions on how and when updates can happen.

These restrictions are reflected in the following principles.

[182] *Flux - Application architecture for building user interfaces*. (https://goo.gl/2dvsNq)
[183] Martin Fowler (July 2011). *CQRS*. (https://goo.gl/mu6Dts)
[184] Martin Fowler (December 2005). *Event Sourcing*. (https://goo.gl/WbJbei)

4.5.1 Principles

Redux can be described in the following fundamental principles:

Single source of truth

The state of your whole application is stored in an object tree within a single store.
A single state tree makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle.
Since all of your state is stored in a single tree, some functionality which has been traditionally difficult to implement - e.g. undo/redo - can become trivial to implement.

State is read-only

The only way to change the state is to emit actions.
Actions are just plain objects that express an intent to transform the state.
Since actions are plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.
Since all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for.

Immutability

The state is immutable. It can be a plain object, an Immutable object, or anything else. You can use any data storage library as long as it supports immutability.

Changes are made with pure functions

To specify how the state tree is transformed by actions, you write pure reducers.
Reducers are just pure functions that take the previous state and an action, and return the next state. Reducers must return new state objects, instead of mutating the previous state.
Reducers can be split off into smaller reducers that manage specific parts of the state tree.
Since reducers are just functions, you can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks.

4.5.2 Concepts

To change something in the state, you need to dispatch an action.
An action is a plain JavaScript object that describes what happened.
Enforcing that every change is described as an action lets you have a clear understanding of what's going on in the app. If something changed, you know why it changed.
To tie state and actions together, reducers takes state and action as arguments, and returns the next state of the app.

Actions

Actions are payloads of information that send data from your application to your store.
They are the only source of information for the store.
You send them to the store using `store.dispatch()`.

For example:

```
{ type: 'ADD_TODO', text: 'This is an action' }
```

```
{ type: 'TOGGLE_TODO', id: 1 }
```

Actions are plain JavaScript objects, with at least the `type` property, that indicates the type of action being performed. Other than type, the structure of an action object is really up to you.

Action creators

Action creators are functions that create actions. They simply return an action. They can also be asynchronous and have side-effects.

For example:

```
function addTodo (text) {
  return { type: 'ADD_TODO', text }
}

function toggleTodo (id) {
  return { type: 'TOGGLE_TODO', id }
}
```

Reducers

Actions only describe what happened, but don't describe how the application's state changes. Reducers specify how the application's state changes in response to actions sent to the store. In Redux, all the application state is stored as a single object. Reducers acts over states. The reducer is a pure function that takes the previous state and an action, and returns the next state.

new_state = *reducer*(*previous_state*, *action*)

A reducer must stays pure. A reducer must never mutate its arguments. Given the same arguments, it should calculate the next state and return it, without side effects.

```
function todoApp (state, action) {
  switch (action.type) {
    case 'ADD_TODO':
      const todo = { text: action.text, completed: false }
      const todos = [...state.todos, todo]
      const new_state = { ...state, todos }
      return new_state

    case 'TOGGLE_TODO':
      const todos = state.todos.map((todo, index) => {
        if (index === action.index) {
          const new_todo = {...todo, completed: !todo.completed }
          return new_todo
        }
        return todo
      })
      const new_state = { ...state, todos }
      return new_state

    default:
      return state
   }
}
```

Note we never write directly to state or its fields, and instead we return new objects.
The fresh todo was constructed using the data from the action.
The new todos is equal to the old todos concatenated with the new todo.
Because we want to update a specific item in the array without resorting to mutations, we have to create a new array with the same items except the item at the index.

Store

The actions represent the facts about "what happened" and the reducers that update the state according to those actions[185].

The Store is the object that brings actions and reducers together:

- it holds application state;
- it allows access to state (via `getState()`)
- it allows state to be updated (via `dispatch(action)`)
- it registers/unregisters listeners (via `subscribe(listener)`)

In a Redux application, there is only a single store. When you want to split your data handling logic, you'll use reducer composition instead of many stores.

Once you have created a store, you can test the update logic even without any UI.

```
// Create the store
const store = createStore(todoApp)

// Log the initial state
console.log(store.getState())

// Every time the state changes, log it
const unsubscribe = store.subscribe(() =>
 console.log(store.getState())
)

// Dispatch some actions
store.dispatch(addTodo('Learn about Redux'))
store.dispatch(addTodo('Learn about CSS frameworks'))
store.dispatch(toggleTodo(0))

// Stop listening to state updates
unsubscribe()
```

[185] *Redux Store*. (https://goo.gl/3j9Vgs)

4.6 Web Components

Components have become a central concept in Web development workflow. Components provide a robust model for architecting and scaling complex applications, allowing for composition from smaller and simpler encapsulated parts.

Modern frameworks like Angular, Vue and Rect have put components at the forefront of development, keeping them as core primitives in their architecture. However, even though component architectures have become more common, arguably the diversity of frameworks and libraries has led to a siloed and fragmented components market. This fragmentation has often kept teams locked into a specific framework, even as times and technologies change.

The desire to tackle this fragmentation, and standardise the web component model has been an ongoing endeavor. Its beginnings sit in the genesis of the "Web Components" specifications circa 2011[186] and were first presented to the world by Alex Russell at Fronteers Conference the same year[187]. The web components specifications grew out of the desire to provide a canonical way of creating components that browsers can understand.

In theory, these specifications and implementations are paving the way for interoperability and composition of components from different vendors. Here we examine the building blocks of Web Components.

The HTML5 Web Components is a set of standard, such as Custom elements, Templates and Shadow DOM[188], to build components.

[186] W3C (November 2011). *Web Components Spec - GitHub repository - first commit.* (https://goo.gl/WCVog7)
[187] Alex Russell. *Web Components and Model Driven Views.* (https://goo.gl/unphU5)
[188] MDN Web Docs. *Web Components.* (https://goo.gl/evYHpG)

4.6.1 Custom Elements

Defining a custom element

To define a custom element, you create a class that extends HTMLElement or one of its subclasses (for example, another custom element), that defines its behavior and public API.

```
class MyComponent extends HTMLElement {
  static get is () { return "my-component" }

  constructor () {
    super()
  }

}
```

```
window.customElements.define("my-component", MyComponent)
```

```
window.customElements.define(MyComponent.is, MyComponent)
```

Custom Element name

By specification, the custom element's name must start with a lower-case ASCII letter and must contain a dash (-). There's also a short list of prohibited element names that match existing names[189].

[189] W3C. *Custom Element*. (https://goo.gl/rs8VpU)

Custom Element constructor

When the element is "upgraded", that is, when it is created or when a previously created element becomes defined, the element constructor is called.

The element constructor has a few special limitations:

- it must call the parameter-less super method;
- it can't include a return statement, unless the return is a simple early return;
- it can't examine the element's attributes or children;
- it can't add any attributes or children to the element.

The HTML5 Custom Element spec provides a set of callbacks called "custom element reactions" that allow you to run user code in response to certain life-cycle changes.

Using a custom element

Using a custom element is like using a standard element.

For example, in HTML:

```
<my-component></my-component>
```

For example, in JavaScript:

```
const myComponent = document.createElement("my-component")
```

```
const myComponent = document.createElement(MyComponent.is)
```

```
const myComponent = new MyComponent();
```

Custom element reactions

A custom element can define special lifecycle hooks for running code during interesting times of its existence. These are called custom element reactions[190]:

- `constructor()` - called when an instance of the element is created or upgraded. Useful for initializing state, settings up event listeners, or creating shadow dom.
- `connectedCallback()` - called every time the element is inserted into the DOM. Useful for running setup code, such as fetching resources or rendering.
- `disconnectedCallback()` - called every time the element is removed from the DOM. Useful for running clean up code.
- `attributeChangedCallback(attrName, oldVal, newVal)` - called when an observed attribute —listed in the `observedAttributes` property— has been added, removed, updated, or replaced, or when an element is created or upgraded
- `adoptedCallback()` - called when the custom element has been moved into a new document (e.g. if `document.adoptNode(element)` is called).

For example:

```
class MyComponent extends HTMLElement {
  static get is () { return "my-component" }
  static get observedAttributes () { return ["color"] }

  constructor () {
    super()
    console.log("The element is created")
  }

  conntectedCallback() {
    console.log("The element is inserted into the DOM")
  }

  disconnectedCallback() {
    console.log("The element is removed from the DOM")
  }

  attributeChangedCallback(attrName, oldVal, newVal) {
    console.log(`Attribute ${attrName} changed from ${oldVal} to ${newVal}`)
  }

}
```

```
window.customElements.define("my-component", MyComponent)
```

[190] Google Developers. *Custom Elements v1: Reusable Web Components - Custom elements reactions.* (https://goo.gl/Xex28W)

Custom elements content

Custom elements can manage their own content by using the DOM APIs inside element code. Reactions come in handy for this.

For example, inserting the following element will produce the following HTML:

```
class MyElement extends HTMLElement {
  static get is () { return 'my-element' }

  connectedCallback() {
    this.innerHTML = "<div>This content was injected</div>";
  }
  …
}
window.customElements.define(MyElement.is, MyElement);
```

```
<my-element>
 <div>This content was injected</div>
</my-element>
```

Overwriting an element's children with new content is generally not a good idea because it's unexpected. Users would be surprised to have their markup thrown out. A better way to add element-defined content is to use shadow DOM.

4.6.2 Templates

The `<template>` element allows you to declare fragments of DOM which are parsed, inert at page load, and can be activated later at runtime. Templates are an ideal placeholder for declaring the structure of a custom element.

```
<template id="custom-template>
     <h1>This is an HTML Template</h1>
</template>
```

```
const template = document.getElementById("custom-template")
const templateContent = template.content
const container = document.getElementById("container")
const templateInstance = templateContent.cloneNode(true)
container.appendChild(templateInstance)
```

4.6.3 Shadow DOM

Shadow DOM provides a way for an element to own, render, and style a chunk of DOM that's separate from the rest of the page.

To use Shadow DOM in a custom element, call this.attachShadow inside the constructor:

```
const template = document.createElement('template')
template.innerHTML = `
  <div>This is the shadow DOM</div>
`

class MyElement extends HTMLElement {

  static get is () { return 'my-element' }

  constructor() {
    super()
    const shadowRoot = this.attachShadow({mode: 'open'})
    const content = template.content.cloneNode(true)
    shadowRoot.appendChild(content)
  }

  …
}
window.customElements.define(MyElement.is, MyElement)
```

```
<my-element>
  #shadow-root
    <div>This is the shadow DOM</div>
</my-element>
```

Shadow DOMs can either be `open` or `closed`: `open` allows access the subtree DOM using `element.shadowRoot` whereas `closed` makes this property return null. Creating a Shadow DOM, in turn, creates a shadow boundary, which alongside encapsulating elements, also encapsulates styles.

4.7 CSS frameworks

Thinking about the relationship between HTML and CSS in terms of “separation of concerns”, could not be the right way to think about HTML and CSS.
Separation of concerns is very black and white, there is or there is not. Thinking about dependency direction, is a more eliciting way to think about HTML and CSS.

There are two ways to write HTML and CSS:

- write CSS that depends on HTML
- write HTML that depends on CSS

That means:

- write restyleable HTML
- write reusable CSS

There is no a “right” approach. There is the best decision made based on what's more important to the specific context of the project you are working on.

Restyleable HTML

Naming elements classes based on the content treats the HTML as a dependency of the CSS.
The HTML is independent, it doesn't care how it looks, it just exposes hooks that the HTML controls.
On the other hand, the CSS is not independent; it needs to know what classes the HTML exposes, and it needs to target those classes to style the HTML.
In this model, the HTML is restyleable, but the CSS is not reusable.

CSS Zen Garden by Dave Shea[191] takes this approach.

[191] Dave Shea. *CSS ZEN GARDEN - The Beauty of CSS Design*. (https://goo.gl/bZvopv)

Reusable CSS

Naming elements classes in a content-agnostic way after the repeating patterns in the UI treats the CSS as a dependency of the HTML.
The CSS is independent, it doesn't care what content it's being applied to, it just exposes a set of building blocks that can be applied to the markup.
The HTML is not independent, it's making use of classes that have been provided by the CSS, and it needs to know what classes exist so that it combine them however it needs to to achieve the desired design.
In this model, the CSS is reusable, but the HTML is not restyleable.

UI frameworks like Bootstrap by Twitter[192] or Foundation by Zurb[193] take this approach.

[192] Twitter. *Bootstrap*. (https://goo.gl/kYfa1W)
[193] ZURB. *Foundation*. (https://goo.gl/9V8kT4)

4.7.1 Bootstrap

Bootstrap is the world's most popular framework for building responsive web sites.

Grid system

Bootstrap's grid system uses a series of containers, rows, and columns to layout and align content. This grid system is built with flexbox[194] and is fully responsive.

For example:

```
<div class="container">
  <div class="row">
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
  </div>
</div>
```

Containers

Containers provide a means to center and horizontally pad your site's contents.
Use `.container` for a responsive pixel width.
Use `.container-fluid` for `width: 100%` across all viewport and device sizes.
Containers are wrappers for rows.

Rows

Rows are wrappers for columns. Each row can contains up to 12 columns.
Only columns may be immediate children of rows.

[194] W3C. *CSS Flexible Box Layout Module Level 1*. (https://goo.gl/3AAdw5)

Columns

Columns are the only place where content should be placed.

Columns have horizontal `padding` (called gutter) for controlling the space between them. The column horizontal `padding` is counteracted on the rows with negative horizontal margins. This way, all the content inside columns is visually aligned down the left side. However, you can remove the margin from rows and the padding from columns with `.no-gutters` on the `.row`.

Column classes indicate the number of columns to use out of the possible 12 per row.
For example:

```
<div class="container">
  <div class="row">
    <div class="col-6"></div>
    <div class="col-4"></div>
    <div class="col-2"></div>
  </div>
</div>
```

Column `widths` are set in percentages, so they're always fluid and sized relative to their parent element. Columns without a specified `width` will automatically layout as equal width columns.

Columns can have different `widths` for different device breakpoints.

Responsive breakpoint

Since Bootstrap is developed to be mobile first, it uses media queries[195] to create sensible breakpoints for layouts and interfaces. These breakpoints are mostly based on minimum viewport widths and allow elements to scale up as the viewport changes.

Bootstrap primarily uses the following media query ranges—or breakpoints— for layout, grid system, and components:

- Extra small <576px: `.col-`
- Small ≥576px: `.col-sm-`
- Medium ≥768px: `.col-md-`
- Large ≥992px: `.col-lg-`
- Extra large ≥1200px: `.col-xl-`

```
/* Extra small devices (portrait phones, less than 576px) */
/* No media query since this is the default in Bootstrap */

/* Small devices (landscape phones, 576px and up) */
@media (min-width: 576px) { ... }

/* Medium devices (tablets, 768px and up) */
@media (min-width: 768px) { ... }

/* Large devices (desktops, 992px and up) */
@media (min-width: 992px) { ... }

/* Extra large devices (large desktops, 1200px and up) */
@media (min-width: 1200px) { ... }
```

Grid breakpoints are based on minimum width media queries, meaning they apply to that one breakpoint and all those above it.

For example, `.col-sm-4` applies to all breakpoints, is equal to `.col-4`;
`.col-sm-4` applies to all breakpoints, but not the first xs breakpoint;
`.col-md-4` applies for medium, large and extra large device; etc.

[195] W3C. *Media Queries*. (https://goo.gl/eC1bxW)

There are also media queries and mixins for targeting a single segment of screen sizes using the minimum and maximum breakpoint widths.

```
/* Extra small devices (portrait phones, less than 576px) */
@media (max-width: 575.98px) { ... }

/* Small devices (landscape phones, 576px and up) */
@media (min-width: 576px) and (max-width: 767.98px) { ... }

/* Medium devices (tablets, 768px and up) */
@media (min-width: 768px) and (max-width: 991.98px) { ... }

/* Large devices (desktops, 992px and up) */
@media (min-width: 992px) and (max-width: 1199.98px) { ... }

/* Extra large devices (large desktops, 1200px and up) */
@media (min-width: 1200px) { ... }
```

For example, the following columns are 50% wide on mobile and 33.3% wide on desktop:

```
<div class="row">
  <div class="col-6 col-md-4"></div>
  <div class="col-6 col-md-4"></div>
  <div class="col-6 col-md-4"></div>
</div>
```

Alignment

To vertically and horizontally align columns, Bootstrap provides flexbox alignment utilities.

Vertical alignment

To vertically align columns in a row, the following classes can be applied to the row:

- `.align-items-start`
- `.align-items-center`
- `.align-items-end`

Otherwise, the following classes can be applied directly to the columns:

- `.align-self-start`
- `.align-self-center`
- `.align-self-end`

Horizontal alignment

To horizontally align columns in a row, the following classes can be applied to the row:

- `.justify-content-start`
- `.justify-content-center`
- `.justify-content-end`
- `.justify-content-around`
- `.justify-content-between`

For example, the following column is vertically and horizontally centered:

```
<div class="row align-items-center justify-content-center">
  <div class="col-4"></div>
</div>
```

Reordering

To control the visual order of a column, use `.order-`*breakpoint*`-`*column* classes.
Where:

- *breakpoint* is one of `xs`, `sm`, `md`, `lg`, `xl`
- *column* is a number between `1` and `12`

There are also responsive `.order-first` and `.order-last` classes that change the order of an element by applying `order: -1` and `order: 13`, respectively.

For example:

```
<div class="container">
  <div class="row">
    <div class="col order-last">First, but last</div>
    <div class="col">Second, but unordered</div>
    <div class="col order-first">Third, but first</div>
  </div>
</div>
```

To move columns to the right, use `.offset-`*breakpoint*`-`*column* classes.
Where:

- *breakpoint* is one of `xs`, `sm`, `md`, `lg`, `xl`
- *column* is a number between `1` and `12`

For example:

```
<div class="row">
  <div class="col-md-4"></div>
  <div class="col-md-4 offset-md-4"></div>
</div>
```

Utilities

Border

To style the border and border-radius of an element, use:

- `border-0` to remove any border
- `border-`*`position`* to add a border at *position*
- `border-`*`position`*`-0` to remove a border at *position*
- `border-`*`color`* to color the border
- `rounded-`*`position`* to add rounded border at *position*
- `rounded-circle` to add circle border
- `rounded-0` to remove rounded border

Where:

- *position* is one of: `top`, `right`, `bottom`, `left`
- *color* is one of: `primary`, `secondary`, `success`, `danger`, `warning`, `info`, `light`, `dark`, `white`

Colors

To change the color, use:

- `text-`*`color`* to change the text color
- `bg-`*`color`* to change the background color

Where:

- *position* is one of: `top`, `right`, `bottom`, `left`
- *color* is one of: `primary`, `secondary`, `success`, `danger`, `warning`, `info`, `light`, `dark`, `white`

Display

To change the value of the display property, use

- `d-`*value* for `xs`
- `d-`*breakpoint*`-`*value* for `sm`, `md`, `lg`, and `xl`.

Where:

- value is one of:
  `none`, `inline`, `inline-block`, `block`,
  `table`, `table-cell`, `table-row`,
  `flex`, `inline-flex`

Components

Alerts

For example:

```
<div class="alert alert-primary" role="alert">Primary</div>
<div class="alert alert-secondary" role="alert">Secondary</div>
<div class="alert alert-success" role="alert">Success</div>
<div class="alert alert-danger" role="alert">Danger</div>
<div class="alert alert-warning" role="alert">Warning</div>
<div class="alert alert-info" role="alert">Info</div>
<div class="alert alert-light" role="alert">Light</div>
<div class="alert alert-dark" role="alert">Dark</div>
```

Badges

For example:

```
<span class="badge badge-primary">Primary</span>
<span class="badge badge-secondary">Secondary</span>
<span class="badge badge-success">Success</span>
<span class="badge badge-danger">Danger</span>
<span class="badge badge-warning">Warning</span>
<span class="badge badge-info">Info</span>
<span class="badge badge-light">Light</span>
<span class="badge badge-dark">Dark</span>
```

Breadcrumb

For example:

```
<nav aria-label="breadcrumb">
  <ol class="breadcrumb">
    <li class="breadcrumb-item"><a href="#">Home</a></li>
    <li class="breadcrumb-item"><a href="#">Library</a></li>
    <li class="breadcrumb-item active" aria-current="page">Data</li>
  </ol>
</nav>
```

Buttons

For example:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```

Card

For example:

```
<div class="card">
  <div class="card-header">Card header</div>
  <img class="card-img-top" src="..." alt="Card image cap">
  <div class="card-body">
    <h5 class="card-title">Card title</h5>
    <h6 class="card-subtitle mb-2 text-muted">Card subtitle</h6>
    <p class="card-text">Card content.</p>
    <a href="#" class="card-link">Card link</a>
    <a href="#" class="card-link">Another link</a>
  </div>
  <div class="card-footer text-muted">Card footer</div>
</div>
```

Nav

```
<ul class="nav">
  <li class="nav-item">
    <a class="nav-link active" href="#">Active</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```

Forms

For example:

```
<form>
  <div class="form-group">
    <label for="email">Email</label>
    <input id="email" type="email" class="form-control" aria-describedby="help">
    <small id="help" class="form-text text-muted">
      We'll never share your email with anyone else.
    </small>
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input id="password" type="password" class="form-control">
  </div>
  <div class="form-check">
    <input id="check" type="checkbox" class="form-check-input" >
    <label for="check" class="form-check-label">Check me out</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

List group

For example:

```
<ul class="list-group">
  <li class="list-group-item">Item</li>
  <li class="list-group-item list-group-item-primary">Primary</li>
  <li class="list-group-item list-group-item-secondary">Secondary</li>
  <li class="list-group-item list-group-item-success">Success</li>
  <li class="list-group-item list-group-item-danger">Danger</li>
  <li class="list-group-item list-group-item-warning">Warning</li>
  <li class="list-group-item list-group-item-info">Info</li>
  <li class="list-group-item list-group-item-light">Light</li>
  <li class="list-group-item list-group-item-dark">Dark</li>
</ul>
```

Pagination

```
<ul class="pagination">
  <li class="page-item"><a class="page-link" href="#">Previous</a></li>
  <li class="page-item"><a class="page-link" href="#">1</a></li>
  <li class="page-item"><a class="page-link" href="#">2</a></li>
  <li class="page-item"><a class="page-link" href="#">3</a></li>
  <li class="page-item"><a class="page-link" href="#">Next</a></li>
</ul>
```

Dynamic components

Other components require a mix of CSS and JavaScript, to enable dynamic behaviours. These components are:

- Carousel
- Collapse
- Modals
- Popovers
- Progress
- Scrollspy
- Tooltips

Bootstrap components depends on the JavaScript library jQuery[196].

[196] jQuerty. *The write less, do more JavaScript library*. (https://goo.gl/vPx9zD)

4.7.2 Atomic CSS

Atomic CSS, like inline styles, offers single-purpose units of style, but applied via classes. This allows for the use of handy things such as media queries, contextual styling and pseudo-classes. The lower specificity of classes also allows for easier overrides. And the short, predictable classnames are highly reusable and compress well.

Display

Single purpose classes for setting the display of an element at any breakpoint.
The `display` property defines box's display type, which consists of the two basic qualities of how an element generates boxes: the inner display type, which defines the kind of formatting context it generates, dictating how its descendant boxes are laid out. the outer display type, which dictates how the box participates in its parent formatting context.[197]

`d`<*modifier*>

Where *modifier* is one of the following:

- `n` = none
- `b` = block
- `ib` = inline-block
- `it` = inline-table
- `t` = table
- `tc` = table-cell
- `t-row` = table-row
- `t-columm` = table-column
- `t-column-group` = table-column-group

Dimensions

Border-box

This module only applies the `border-box` model to certain elements.
The benefit of `border-box` as opposed to `content-box` (which is the default per the CSS spec) is that when you declare a width, that is the width of the element, regardless of how much border or padding you add to the element.

[197] W3C. *CSS Display Module Level 3*. (https://goo.gl/T5kmqy)

Widths

The widths module contains both a five-step width scale based on powers of two as well as a series of percentage values that can be combined with floats for an infinitely nestable and fully responsive grid system.

`w`<*modifier*>

Where *modifier* is one of the following:

- `1`, `2`, `3`, `4`, `5` = from 1st to 5th step in width scale
- `-10`, -20, -25, -30, -33, -34, -40, -50, -60, -70, -75, -80, -90, -100 = percentage
- `-third`, `-two-thirds` = fraction
- `-auto` = string value auto

Max widths

The max-widths module contains both a ten-step scale based on powers of two as well as the ability to constrain element widths to a maximum of 100%.
Max widths can be combined with widths to ensure that your content doesn't get too wide on larger monitors. Max-widths can also help keep embedded media within the canvas.

`mw`<*modifier*>

Where *modifier* is one of the following:

- `1` to `9` = 1st to 9th step in width scale
- `-100` = literal value %
- `-none` = none

Heights

The heights module contains both a five-step height scale based on powers of two as well as a series of percentage values. Explicit values can be used inside of any parent. Percentage values will only work inside of a parent element that has a declared height.

`h`<*modifier*>

Where *modifier* is one of the following:

- `1` to `5` = from 1st to 5th step in height scale
- `-25`, `-50`, `-75`, `-100` = percentage
- `-auto` = string value auto

Spacing

Spacing comes in two flavors. Depending on borders and background colors, the difference between padding and margin can be invisible to the user, but not to the designer.

Good design is based on math. Certain patterns and ratios are so prevalent in nature and music that they can't be denied as elegant design solutions. Even in the 18th century, pages in books were designed with ratios. In the 21st century, we have gotten away from this on the web, often using magic numbers to match a 'spec' that has been produced in a graphics program such as photoshop, illustrator, or sketch. While these programs are useful for sketching ideas, they are not 100% accurate in their reflection of how the web works across device sizes or how things get drawn to the screen.

Tachyons features a spacing scale based on powers of two that starts at .25rem (for most devices this will be the equivalent of 4px). Since tachyons uses rem units with px as a fallback, if a user has declared a different base font-size for their device, your spacing will scale based on a defined ratio that has stood the test of time. As powers of two will always produce integers, there will be no problems with sub pixel rendering across browsers. Computers aren't that great at math and so decimals lead to inconsistencies across platforms. Inconsistencies should be avoided where possible. You'll find that when using a well thought out scale - things just line up. It works, with little effort, regardless of your design knowledge or sensibilities.

Margin and padding

An eight step powers of two scale ranging from 0 to 16rem.

<base><modifier><size>

Where:

- *base* is one of the following:
    - `p` = padding
    - `m` = margin
- *modifiers* is one of the following:
    - `a` = all
    - `h` = horizontal
    - `v` = vertical
    - `t` = top
    - `r` = right
    - `b` = bottom
    - `l` = left
- *size* is an integer from 0 (none) to 7 (7th step in spacing scale)

For example:

- `pa0` = no padding
- `ph1` = one space unit horizontal padding
- `pv2` = 2 space units vertical padding
- `pl3` = 3 space units left padding
- `mr4` = 4 space units right margin
- `mt5` = 5 space units top margin
- `mb7` = 7 space units bottom margin

Flexbox

You can combine display, float, padding, and widths to construct a wide variety of grids, or use the flexbox module.

Flex

The flexbox module provide the following classes:

- `flex` to display child elements into a single row.
- `flex-wrap` to wrap child elements to multiple rows if they overflow the parent.
- `flex-wrap-reverse` like `flex-wrap` but in reverse ordering.
- `flex-column` to display child elements into a single column.
- `flex-column-reverse` like `flex-column` but in reverse ordering.

Align items

To align items in a flex container:

- `items-center` to pack items from the center
- `items-start` to pack items from the start
- `items-end` to pack items from the end

Justify content

To justify content in a flex container:

- `center` to justify items from the center
- `between` to equally divide free space between items
- `around` to equally divide free space around items

4.7.3 Maintainable CSS

“Maintainable CSS” is an approach, introduced by Adam Silver, to writing modular, scalable, maintainable CSS[198].
It is not a library or framework. It is a set of principles and conventions to help you write CSS for small or large-scale websites.

Maintainable means that you can make styling changes without worrying about accidentally causing problems elsewhere.
Scalable means that as CSS increases in size, it’s still easy to maintain.
Modular means that independent units that can be combined with other modules to form a more complex structure.

Semantics

Semantic HTML is not only about the elements you use, but mainly for the names you chose for the element classes. Naming is the most important aspect of writing maintainable CSS.
There are two main approaches: the semantic approach and the non-semantic approach.

Non-semantic classes give you an idea of what an element looks like.
They do not convey what an element represents.

For example:

```
<div class="border-0">
<div class="pull-left">
<div class="col-xs-4">
```

Semantic classes give you an idea of what an element represents.
They do not convey the style of an element.

For example:

```
<div class="basket">
<div class="product">
<div class="searchResults">
```

[198] Adam Silver. *Maintainable CSS*. (https://goo.gl/RSk6Um)

Semantic classes are readable

Non-semantic classes usually are abbreviations. Abbreviations are hard to read. It's easier to read words than abbreviations. Abbreviations have to be broken down and mapped cognitively, assuming we know what they mean in the first place.
However, even when not abbreviated, you need to wade through many classes to work out what's happening; which classes override which; which apply at certain breakpoint; etc.
The content is obfuscated by the surrounding HTML.

For example:

```
<div class="pb3 pb4-ns pt4 pt5-ns mt4 black-70 fl-l w-50-l">
  <h1 class="f4 fw6 f1-ns lh-title measure mt0">Heading</h1>
  <p class="f5 f4-ns fw4 b measure dib-m lh-copy">Tagline</p>
</div>
```

Semantic classes are easy to read. They do not require any mental mapping.
The content is no longer obfuscated. We know where the content begins and ends.
The CSS is easy to read because it has dedicated language constructs that exist for this purpose already.

For example:

```
<div class="hero">
  <h1 class="hero-title">Heading</h1>
  <p class="hero-tagline">Tagline</p>
</div>
```

Semantic class ease building responsive sites

To make a column system actually responsive (a-la bootstrap) we would need to introduce a naming convention such as *col-xs-*, *col-md-*, *col-lg-*, *col-xl-*, to disambiguate the cases.
This means recreates language constructs already found and standardised in CSS. At certain breakpoints, the classes are misleading and redundant.

Semantic classes are encapsulated to the module's design and content.
It's easy to style elements without having to write a multitude of classes and changing the HTML again. These classes are meaningful in small and big screens.
Media queries can be used to clear elements only when needed.

Semantic class are easier to find and to debug

Since semantic classes are unique, a search yields only one result, making it easy to track down the HTML. Searching for HTML with a non-semantic class yields many results.
Inspecting an element with a multitude of atomic classes, means wading through many selectors. With a semantic class, there's only one, making it far easier to work with.

Semantic class eliminate the risk of regression

Updating a visual class could cause regression across a multitude of elements. Updating a semantic class only applies to the module in question, eliminating regression altogether.

Semantic class provide hooks for automated tests and JavaScript

Automated functional tests work by searching for, and interacting with elements, like clicking a link, finding a text box, typing in text, submitting a form, verifying some criteria, etc.
We can't use non-semantic classes to target specific elements. And adding hooks specifically for tests is wasteful as the user has to download this stuff.
We can't use non-semantic classes to target specific elements in order to enhance them with Javascript as well.

Semantic class don't need maintaining

If we name a thing based on what it is, we won't have to update the HTML again e.g. a heading is always a heading, no matter what it looks like.
With visual classes, both the HTML and the CSS need updating (assuming there aren't any selectors available for use).

Semantic class are recommended by the standards

Last but not least, semantic class are recommended by the W3C HTML5 Standard.
On using the class attribute, HTML5 Specs say in 3.2.3.6[199]:

> *Authors are encouraged to use values that describe the nature of the content, rather than values that describe the desired presentation of the content.*

---

[199] W3C. HTML5 Spec. *Elements*. (https://goo.gl/dsGSwo)

Reuse

"Do not Repeat Yourself" (or DRY) is a principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions, or repetition of the same data, using data normalization to avoid redundancy.
As Harry Roberts says:

> *DRY is often misinterpreted as the necessity to never repeat the exact same thing twice. This is impractical and usually counterproductive, and can lead to forced abstractions, over-thought and over-engineered code.*

This forced abstraction, over-thought and over-engineered code often results in visual and atomic classes.

In CSS, you can apply a set of rules to more than one selector.
Using a post-processor like SASS you do this using @extend.
For example:

```
.some-thing,
.another-thing {
  /* shared rules */
}
```

This approach should be used for convenience, not for performance.
If the abstraction only has one rule, we're simply exchanging one line of code for another.
If a selector deviates from the rules inside the abstraction, it should be removed from the comma-separated list of selectors for that abstraction. Otherwise it could regress the other selectors and cause override issues.

We often overthink performance and get obsessed with tiny details. Even if CSS did total more than 100kb, there's little to gain from mindlessly striving for DRYness.
Making CSS small makes HTML big. CSS can always be cached. But HTML often contains dynamic and personalised content—so it can't be cached.

Striving for DRY leads to over-thought and over-engineered code. In doing so we make maintenance harder, not easier. Instead of obsessing over styles, we should focus on reusing tangible modules.

Conventions

Maintainable CSS has the following convention:

```
.<module>[-<component>][-<state>] {}
```

Where:

- square brackets are optional depending on the module.

For example:

```
/* Module container */
.searchResults {}

/* Component */
.searchResults-heading {}

/* State */
.searchResults-isLoading {}
```

Where:

- component and state are both delimited by a dash
- names are written with lowerCamelCase
- selectors are prefixed with the module name

Modules

A module is a distinct, independent unit, that can be combined with other modules to form a more complex structure. If one of the units is taken away, the others still work.
For example, in a blog site, header, navigation, footer, post, post list, can all be considered to be modules.

A module is made up of components.
For example, a logo module might consist of copy, an image and a link, each of which are components.

A module, without the components, is incomplete or broken.
For example, the logo component, without the image is broken, without the link is incomplete.

Sometimes it's hard to decide whether something should be a component or a module. There is no a strict rule. It depends by the context.

A module, by definition, is a reusable chunk of HTML and CSS. Before a group of elements can be upgraded into a module, we must first understand what it is and what its different use cases are.

Only then, can we design the right abstraction. And in doing so, we avoid complexity at the same time, which is the source of unmaintainable CSS.

State

Quite often, particularly with richer user interfaces, styling needs to be applied in response to an element's change of state.

For example, a module (or component) could be represented in the following states:

- showing or hiding
- active or inactive
- disabled or enabled
- loading or loaded
- isEmpty or isFull

To represent state we need an additional class which should be added to the module (or component) element to which it pertains.

The class name is prefixed with the module (or component) because whilst states might be common, associated styles might not.

For example:

```
<div class="card card-isEmpty">
```

There are two approaches of reusing state:
encapsulating state to the module or creating a global state class.
For example, encapsulating state to the module, the CSS could be:

```
.moduleA-isHidden,
.moduleB-isHidden {
  display: none;
}
```

The trade-off is that this list could grow quickly.
And every time you add behavior, you need to update the CSS.
For example, creating a global state class, the CSS could be:

```
.globalState-isHidden {
  display: none;
}
```

Thinking about state requires to consider how this state affects behaviour as well as style. Different components may share the same behaviour, but they may look rather different.

Modifiers

Like state, modifiers also override styles. They are useful when modules (or components) have small and well understood differences. Because the differences are small and well understood, this type of reuse is more maintainable.

For example, most sites have a primary and secondary button style. If all that changes is one or two styles we can have a modifier for primary and secondary buttons as follows:

```
.button {
  /* shared button style */
}

.button--primary {
  /* primary button style */
}

.button--secondary {
  /* secondary button style */
}
```

4.7.4 Styled Components

Styled-components utilises tagged template literals to style your components.
It removes the mapping between components and styles. This means that when you're defining your styles, you're actually creating a normal React component, that has your styles attached to it.

For example:

```
// Create a Title component that'll render an <h1> tag with some styles
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`

// Create a Wrapper component that'll render a <section> tag with some styles
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`

// Use styled Title and Wrapper like any other React component
render(
  <Wrapper>
    <Title>
      Hello Styled Components!
    </Title>
  </Wrapper>
)
```

Interpolation

You can pass a function ("interpolations") to a styled component's template literal to adapt it based on its props.

For example:

```
const Button = styled.button`
  background: ${props => props.primary ? 'palevioletred' : 'white'};
  color: ${props => props.primary ? 'white' : 'palevioletred'};
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`


render(
  <div>
    <Button>Normal</Button>
    <Button primary>Primary</Button>
  </div>
)
```

Styling any components

The styled method works perfectly on all of your own or any third-party components as well, as long as they pass the className prop to their rendered sub-components, which should pass it too, and so on. Ultimately, the className must be passed down the line to an actual DOM node for the styling to take any effect.

If you're using any external library, you can consider using this pattern to turn them into styled components. The same pattern works for your own components as well, if you need some components to stay unstyled on their own.

For example:

```
const Link = ({ className, children }) => (
  <a className={className}>
    {children}
  </a>
)

const StyledLink = styled(Link)`
  color: palevioletred;
  font-weight: bold;
`;

render(
  <div>
    <Link>Unstyled, boring Link</Link>
    <br />
    <StyledLink>Styled, exciting Link</StyledLink>
  </div>
);
```

Extending Styles

Quite frequently you might want to use a component, but change it slightly for a single case. Now you could pass in an interpolated function and change them based on some props, but that's quite a lot of effort for overriding the styles once.

To do this in an easier way you can call extend on the component to generate another. You style it like any other styled component. It overrides duplicate styles from the initial component and keeps the others around.

```
// The Button from the last section without the interpolations
const Button = styled.button`
  color: palevioletred;
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`;

// We're extending Button with some extra styles
const TomatoButton = Button.extend`
  color: tomato;
  border-color: tomato;
`;

render(
  <div>
    <Button>Normal Button</Button>
    <TomatoButton>Tomato Button</TomatoButton>
  </div>
);
```

4.7.5 Styling Web Components

Shadow DOM is a spec that gives you DOM and style encapsulation. This is great for reusable web components, as it reduces the ability of these components' styles getting accidentally overwritten.

When talking about styling a component, there are usually two different problems you might want to solve:

- styling a third-party element;
- theming a set of third-party elements.

There have been several previous attempts at solving this, some more successful than others.

:shadow and /deep/

First came :shadow and `/deep/` (which have since been deprecated, and removed as of Chrome 60). These were shadow-piercing selectors that allowed you to target any node in an element's Shadow DOM. Apart from slowing performance, they also required the user of an element to be intimately familiar with some random element's implementation, which was unlikely and lead to them just breaking the whole element by accident.

Custom properties

Custom properties allow you to create custom CSS properties that can be used throughout a document. In particular, they pierce the shadow boundary, which means they can be used for styling elements with a Shadow DOM.

For example:

```
:root {
  --primary-color: coral;
}
```

```
element {
  background-color: var(--primary-color);
}
```

@apply rule

The problem with using just custom properties for styling/theming is that it places the onus on the element author to basically declare every possible styleable property as a custom property.

As a result, `@apply` was proposed, which basically allowed a custom property to hold an entire ruleset.

For example:

```
/* outer page */
x-component {
  --heading-style: {
    color: red;
    text-decoration: underline;
    font-weight: bold;
  };
}

/* shadow DOM stylesheet */
.heading {
  @apply(--heading-style);
}
```

However, this approach was abandoned, since it interacted pretty poorly with pseudo classes and elements (like `:focus`, `:hover`, `::placeholder` for `input`), which still meant the element author would have to define more combination of these properties to be used in the right places.

::part and ::theme pseudo-elements

The current new W3C proposal is ::part and ::theme[200], a set of pseudo-elements that allow you to style inside a shadow tree, from outside of that shadow tree.
Unlike :shadow and /deep/, ::part and ::theme do not allow you to style arbitrary elements inside a shadow tree: they only allow you to style elements that an author has tagged as being eligible for styling.

::part pseudo-element

You can specify a "styleable" part on any element in your shadow tree.
For example:

```
<x-element>
  #shadow-root
    <div part="some-box">
      <span></span> /* not styleable */
    </div>
    <input part="some-input">
    <div></div> /* not styleable */
</x-element>
```

If you're in a document that has an <x-foo> in it, then you can style those parts with:

```
x-element::part(some-box) { … }
```

You can use other pseudo elements or selectors (that were not explicitly exposed as shadow parts). For example:

```
x-element::part(some-box):hover { … }
x-element::part(some-input)::placeholder { … }
```

You cannot select inside of those parts. For example, the followings do not work:

```
x-element::part(some-box) span { … }
x-element::part(some-box)::part(some-other-thing) { … }
```

[200] W3C. *CSS Shadow Parts*. (https://goo.gl/1hpxnc)

Forwarding parts

You cannot style this part more than one level up if you do not forward it.
So without any extra work, if you have an element that contains the x-element like this:

```
<x-bar>
  #shadow-root
    <x-element></x-element>
</x-bar>
```

You cannot select and style the the part like this:

```
x-element::part(some-box) { ... }
```

However, you can explicitly forward a child's part to be styleable outside of the parent's shadow tree. So to allow the some-box part to be styleable by x-element's parent, it would have to be exposed:

```
<x-bar>
  #shadow-root
    <x-element part="* => bar-*"></x-element>
</x-bar>
```

You can explicitly forward x-foo's parts that you know about (i.e. some-box and some-input) as they are. For example:

```
<x-foo part="some-box => some-box, some-input => some-input"></x-foo>
```

So, these selectors would match:

```
x-bar::part(some-box) { … }
x-bar::part(some-input) { … }
```

Rename forwarding

You can explicitly forward (some) of x-element's parts (i.e. some-input) but rename them.

```
<x-bar>
  #shadow-root
    <x-element part="some-input => element-input"></x-element>
</x-bar>
```

These selectors would match:

```
x-bar::part(element-input) { … }
```

These selectors would not match:

```
x-bar::part(some-box) { ... }
x-bar::part(some-input) { ... }
```

Prefix forwarding

You can implicitly forward all of x-element's parts as they are, but prefixed.

```
<x-element part="* => element-*"></x-element>
```

These selectors would match:

```
x-bar::part(element-some-box) { … }
x-bar::part(element-some-input) { … }
```

These selectors would not match:

```
x-bar::part(some-box) { … }
x-bar::part(some-input) { … }
```

You can chain these, as well as add a part to x-foo itself.
All of these are valid:

```
<x-bar>
  #shadow-root
    <x-element part="some-foo, * => bar-*"></x-foo>
</x-bar>
```

```
<x-bar>
  #shadow-root
    <x-element part="some-foo, some-input => bar-input"></x-element>
</x-bar>
```

You cannot forward all parts at once, i.e. part="* => *" since this might break your element in the future (if the nested shadow element adds new parts). So this is invalid:

```
<x-form>
  #shadow-root
    <x-bar part="* => *">
      #shadow-root
        <x-element part="* => *"></x-element>
    </x-bar>
</x-form>
```

However, you can forward all the parts if you prefix them. So this is valid:

```
<x-form>
  #shadow-root
    <x-bar part="* => bar-*">
      #shadow-root
        <x-element part="* => foo-*"></x-element>
    </x-bar>
</x-form>
```

This selector would be valid:

```
x-form::part(bar-foo-some-input) { ... }
```

::theme pseudo-element

Given the above prefixing rules, to style all inputs in a document at once, you need to Ensure that all elements correctly forward their parts and Select all their parts.

So given this shadow tree:

```
<submit-form>
  #shadow-root
    <x-form part="some-input => some-input, some-box => some-box">
      #shadow-root
        <x-bar part="some-input => some-input, some-box => some-box">
          #shadow-root
            <x-foo part="some-input => some-input, some-box => some-box"></x-foo>
        </x-bar>
    </x-form>
</submit-form>
```

```
<x-form></x-form>
<x-bar></x-bar>
```

You can style all the inputs with:

```
:root::part(some-input) { ... }
```

This is a lot of effort on the element author, but easy on the theme user.

If you hadn’t forwarded them with the same name and some-input was used at every level of the app (the non contrived example is just an <a> tag that’s used in many shadow roots), then you’d have to write:

```
:root::part(form-bar-foo-some-input),
:root::part(bar-foo-some-input,
:root::part(foo-some-input),
:root::part(some-input) { ... }
```

This is a lot of effort on the theme user, but easy on the element author.

Both of these examples show that if an element author forgot to forward a part, then the app can’t be themed correctly.

`::theme` matches any parts with that name, anywhere in the document.
This means that if you hadn't forwarded any parts, i.e.:

```
<x-bar>
  #shadow-root
    <x-foo></x-foo>
    <x-foo></x-foo>
    <x-foo></x-foo>
</x-bar>
```

You could style all of the inputs in x-bar with:

```
x-bar::theme(some-input) { … }
```

This can go arbitrarily deep in the shadow tree. So, no matter how deeply nested they are, you could style all the inputs with part="some-input" in the app with:

```
:root::theme(some-input) { … }
```

CHAPTER 5
DESIDÈRA
THE CONTENT MANAGEMENT SYSTEM FOR THE DECENTRALIZED WEB

In this chapter, a novel approach to design websites for the Decentralized Web is presented. In particular, a novel Content Management System over a content-addressed peer-to-peer file system, called Desidèra, is presented.

5.1 Motivation

The systems used to build and deploy websites onto the Web, from the traditional Web Content Management Systems to the cutting-edge static progressive web app generators, have not been thought nor ready for building and deploying websites onto a content-addressed peer-to-peer network.

Web Content Management Systems

Traditional Web CMSs introduced the “separation of content from presentation” by means a templating system, as well as a flourishing ecosystem of themes and plugins.
Any software platform that aims to be widely used should provide a way to be customized, extended, and to allow the community of developers to contribute to the rise of the platform.
For this reason, a Web CMS that aims to be widely used should provide and nourish a themes and plugins ecosystem.

Traditional Web CMSs essentially dynamically generate web pages at request. When a user visits a web page, expecting the latest content, the web server queries the database to get the content, pass the results to the templating engine, that compose the HTML, and let the server serve the dynamically generated page.

Traditional Web CMSs do not fit well in a peer-to-peer network, because the server that generates the web pages would introduce a singular point-of-failure in the distributed system, that is the node dedicated to generate the pages.

Static Site Generators

Static site generators shift the heavy load from the moment users request the webpage to the moment the webpage actually changes, when the website is updated, generating a structure of purely static HTML files that are ready to be delivered as are to the users.

Static site generators seem to have been becoming more and more popular, but they're not one of those ephemeral novelty things that grow in popularity as quickly as they fall into oblivion shortly after.

Static site generators produce static website that can be delivered by a Content Delivery Network, and even by peer-to-peer network.

Static site generators seems to fit well in a peer-to-peer network. However they are not optimized for that.

Static Progressive Web App generators allow to build the Single Page Applications, or Web Apps, so that they can be statically served. These systems combine the advantages of decoupled systems and static site generators with the potential of a web app.

In a traditional static generated website, any change —to the content, or to the theme— requires the system to rebuild the entire website. If the change affected the content of one page, the system will build that web page. But, if the change affected the theme, the system will build the entire web site.

In a distributed peer-to-peer network, having new pages on every change don't let to take advantage of the sharing fundamental aspect of the distributed network.

Decoupled and Headless Web CMSs

Decoupled software architectures come with a wide range of advantages and promises. They are better at fulfilling the requirements for flexibility and for agility as well as the ever-growing need for scalability. This trend applies to all sort of software, especially for Web architectures and systems.

A traditional Web CMS is essentially composed by three fundamental components: the content management, the content delivery and the content presentation component.

A Decoupled CMS is essentially a regular Web CMS where the content management component is independent of, decoupled from, the content delivery and content presentation components.

A Headless CMS is essentially a Decoupled CMS where the content presentation component is not present. For this reason, they are called headless, without the head.

While traditional CMSs have to be hosted and built together with the website every time it is served, a Headless CMS does not care where it is serving its content to, since it is no longer attached to the frontend.

Headless CMSs introduced the “data-driven” content developing. Content is structured and defined by data structures, or model types. Headless CMSs provide the infrastructure to design and build the data, usually called models, that is the representation of the content of the website.

Headless CMSs give the web designer the honors and burdens to build the front-end of the website. Usually a Single Page Application is built, by means a Front-End Web Framework.

Desidera

Desidera is a a novel Content Management System over a content-addressed peer-to-peer file system, i.e. the InterPlanetary File System.

Why the name

From latin

- *de-*, the privative particle which means from or away from, to indicate the place from which someone or something departs or withdraws.
- *sidèra*, the plural of *sidus*, which means star, constellation.

Desidèra means desire.

Etymologically, “to desire” means to be far from the stars, to be deprived of them. Desire is therefore disorientation, lack of precise references.
It seems a paradox. And yet, precisely that “lack of stars” turns into yearning, in research, when you feel lost and the difficulty of living can push you to find yourself.
Then, when curiosity rises, the desire becomes a precious gift. It becomes the yearning that guides us to travel the distance between us and our stars. The force that moves our steps.

Desidera as the desire to discover, to know, to share the knowledge.

5.2 Principles

Desidera promotes the content-first design approach, firmly pursuing the "separation of concerns" principle.

Design content at first

Content is why a website exists, it is what gives a website meaning, it is why people visit a website, it is what keeps people engaged.

Content should be at the core of the design process of a website, drawing people in, and allowing them to understand, a website's purpose and take action. Without any sort of structured and organized content, building a website would be a more time-consuming process, visiting a website would be a frustrating experience.

The content-first design approach allows you present the information in an organized way. This focus allows the design to highlight the information and lead to an improved user experience.

The content-first design approach brings together designers and content creators. Requirements and limitations in either area will affect the work for each of these roles.

Content creators and designers are empowered to produce stronger work when they share a common goal and understand how all the pieces work together[201].

---

[201] Jeff Cardello (September 2017). *The modern web design process: putting content first.* (https://goo.gl/mVuJ8P)

Design content by data

Information is large, complex, and rife with relationships that are important to its meaning but impossible for a computer to decipher.

Dealing with data is much easier than dealing with information: data is small, simple, and all its relationships are clearly known or else ignored.

Content is a compromise between the usefulness of data and the richness of information: content is a digestible form of information, it is rich information wrapped in simple data.

For the “content compromise”, the problem of information management is simplified, creating a set of data that represents the best guess of the important aspects of the information to manage, and then, using the data capabilities of the information management system to manage the information via the simplified data.

5.3 Data

5.3.1 Objects

An Object has three properties:

- meta: contains metadata;
- data: contains the data;
- links: contains links to other objects.

A meta object has three properties:

- version: is the version number for the versioning system;
- timestamp: is the date time the object was created at;
- type: it the type of the data for reading the data correctly.

Data depends on the type reported in the meta object.

A link has three properties:

- name: is the name of the object the link point to;
- size: is the size of the subtree;
- hash: is the hash of the object.

Objects have no name. Name of an object is given by the link that points to the object.
The same object can have different names, depending on the other objects that point to it.

Objects define a directed graph over a directed acyclic graph.

5.3.2 Paths

Object links traversal define paths. An object is identified by the path - that is the list of links - to traverse to get the object.

For example:

```
root = { meta, data, [{ name: "a", hash: "#a" }]  }
#a = { meta, data, [{ name: "b", hash: "#b" }] }
#b = { meta, data, [{ name: "b", hash: "#b" }] }
```

The object returned by traversing the path “`/a/b`” is the object identified by the hash `#b`.

5.3.3 API

There are just two methods:

- `set(path, meta, data)`: returns the hash of the object created with `meta` and `data`.
- `get(path, timestamp)`: returns the object with the given hash, created at (or before) the given timestamp.
- `delete(path)`: return the hash of the deleted object

Since data is immutable, an update to data is a new object.
This preserves the versioning.

5.4 Types

5.4.1 Content types

Each content type contain basic information, such as:

- `name` - the name of the content type.
- `description` - the description of the content type.
- `properties` - a list of fields of the content type.
- `controls` - a list of widget used to display the fields.

Properties

Each property contain basic information, such as:

- name - the name of the property
- description - the description of the property
- type - the type of the property
- metadata depending of the type

A property type can be one of the following:

- `String` - a short text (e.g. for titles and names).
- `Text` - a long text (e.g. for paragraphs of text).
- `Number` - a decimal number.
- `Date` - a date and time (in ISO 8601 format).
- `Location` - a coordinate object (latitude and longitude of a location).
- `Boolean` - a value that has two states (e.g. yes/no or true/false).
- `Media` - a link to an asset.
- `Link` - a link to another entry.
- `Array` - a list values.
- `Object` - a set of key/value pairs.

Controls

Each property can be edited by a widget.
Controls is a list of property/widget pairings.

For example, in the Blog Post type:

```
{
  …
  "properties": [
    { "id": "title", "name": "Title", "type": "Text" },
    { "id": "body", "name": "Body", "type": "Text" },
    { "id": "category", "name": "Category", "type": "Symbol" }
  ],
  "controls": [
    { "property_id": "title", "widget": "single_line" },
    { "property_id": "body", "widget": "multiple_line" },
    { "property_id": "category", "widget": "dropdown" }
  ]
}
```

Where:

- the `title` field is rendered as an input field (using the `single_line` widget)
- the `body` is rendered as a normal text area (using the `multiple_line` widget)
- the `category` is rendered as a dropdown field (using the `dropdown` widget)

Applicable widgets per field type

There are sets of applicable widgets per content type field type.

For the `Asset` type:

- `asset_link` - search, attach, and preview an asset.
- `asset_links` - search, attach, reorder, and preview multiple assets.
- `asset_gallery` - search, attach, reorder, preview multiple assets in a gallery

For the `Boolen` type:

- `boolean` - radio buttons with customizable labels.

For the `Date` type:

- `date_picker` - select date, time, and timezone.

For the `Entry` type:

- `entry_link` - search and attach another entry.
- `entry_links` - search and attach multiple entries.
- `entry_card` - search, attach, and preview another entry.
- `entry_cards` - search, attach and preview multiple entries.

For the `Number` type:

- `number` - simple input for numbers.
- `rating` - uses stars to select a number.

For the `Location` type:

- `location` - a map to select or find coordinates from an address.

For the `Object` type:

- `object` - a code editor for JSON.

For the `Symbol` type:

- `url` - a text input that also shows a preview of the given URL.
- `slug` - generates a slug and validates its uniqueness across entries.
- `list_input` - text input that splits values on , and stores them as an array.
- `checkbox` - a group of checkboxes.
- `tag` - a text input to add a string to the list.

For the `Text` type:

- `single_line` - a simple text input field.
- `multiple_line` - a simple textarea input.
- `dropdown` - a select element.
- `markdown` - a full-fledged markdown editor.
- `radio` - a group of radio buttons.

Widget settings

You can pass custom settings to a control that change the behavior or presentation of a widget.

For example, the entry for a field of type Boolean would look like this:

```
{
  "property_id": "is_draft",
  "widget_id": "boolean",
  "settings": {
    "help_text": "Is the post in draft?",
    "true_label": "yes",
    "false_label": "no",
  }
}
```

Where:

- `help_text` shows extra information with the widget.
- `true_label` shows this text next to the radio button that sets this value to “true”.
- `false_label` shows this text next to the radio button that sets this value to “false”.

5.5 Content

5.5.1 Organizing content

When confronted with a new and complex information system, users build mental models. They use these models to assess relations among topics and to guess where to find things they haven't seen before.

The success of the organization of a website will be determined largely by how well the website's information architecture matches the users' expectations. A logical, consistently named site organization allows users to make successful predictions about where to find things. Consistent methods of organizing and displaying information permit users to extend their knowledge from familiar pages to unfamiliar ones.

If a website mislead users with a structure that is neither logical nor predictable, or constantly uses different or ambiguous terms to describe site features, users will be frustrated by the difficulties of getting around and understanding what the website has to offer. Organized content leads to enjoyable website.

A website is organized into sections, and each section contains pages of the same type.
For example, the folder posts contains only pages (folders) of same type (e.g. post).

For example, the following content organize pages and posts:

```
content
├── pages
|   ├── page_1
|   |   ├── meta
|   |   └── content
|   └── page_2
|       ├── meta
|       └── content
└── posts
    ├── post_1
    |   ├── meta
    |   └── content
    └── post_2
        ├── meta
        └── index
```

5.5.2 Page type

The Page type represents a web page.

The meta of a page includes the following fields:

- `type` - the page type
- `title` - the title for the content
- `description` - the description for the content
- `keywords` - the meta keywords for the content
- `draft` - if true, the content will not be rendered
- `priority` - the priority to order the content
- `data` - data specific to the content type

A page can be of different type (specified it the type field).
Any type of content could have a specific set of metadata (specified in the data field).

The `data` object depends on the type of the page.

In a static site generator, page metadata are stored within the page, in the "frontmatter", in the head of the document (separated from the content with a special sequence of characters).
Here, metadata and content are stored in separated files.
Separating content and metadata allows to modify them independently.
Metadata files could be written in JSON, as well as YAML or TOML.
Desidera detect the type of the file and use the appropriate parser to get the data.

5.5.3 Resource type

Any resource used in a page could have metadata associated with it. As for the page type, metadata for a resource are kept separate from the resource.

The metadata of a file include the following fields:

- `name` - the file name
- `type` - the type of file

5.5.4 Post type

The Post type represents a blog post.

The meta of a post includes the following fields:

- `title` - the title for the content
- `description` - the description for the content
- `keywords` - the meta keywords for the content
- `draft` - if true, the content will not be rendered
- `author` - the ref to the author
- `created_at` - the timestamp of creation
- `published_at` - the timestamp of publishing
- `excerpt` - the excerpt

5.6 Theme

The theme contains the code that makes the content live.

5.6.1 Components

Desidera doesn't force to use a particular frontend framework, but promotes the use of Web standards, such as HTML5 Web Components.

Web Components are a set of HTML5 APIs to create new custom, reusable, encapsulated HTML tags to be used in web pages. Web Components work across modern browsers, and can be used with any JavaScript library or framework that works with HTML5.

Websites could be as complex as any other software applications. It's essential to find the right way to divide up the development work with minimal overlap between systems in order to be more efficient. Componentization (in general) is how this is done. Any component system should reduce overall complexity by providing isolation, or a natural barrier that hides the complexity of one system from another. Good isolation also makes reusability and serviceability easier[202].

In a Desidera theme, "*everything is a component*".
A theme must contain at least one component: the default component for pages.

There are components to build different types of page (by convention, the page-* components) and to build different parts within a page (by convention, the part-* components), such as the header (part-header), the footer (part-footer), the navigation menu (part-nav).

[202] T. Leithead and A. Eicholz. (July 2015). *Bringing componentization to the web: An overview of Web Components*. (https://goo.gl/6A49af)

Page components

In Desidera, a page-component is a top-level component.
A page-component represents a page, and it is composed by part-components.
A page-component should extend the provided Page class.

The Page class exposes the following methods/life-cycle hooks:

- `ready` - when the page is ready to be rendered.
- `connected` - when the component is mounted to the app-container.
- `disconnected` - when the page is unmounted from the app-container.

A page-component, once mounted to the app-container, has access to a data object, that includes at least the following fields:

- `path` - the path of the page
- `route` - the route to access the page
- `meta` - the meta data object of the page
- `children` - the list of subfolders in the page folder
- `files` - the list of files in the page folder

Part components

Desidera default theme provides a built-in set of components, focused to design content-driven websites, such as: page-post to render a page for a single post, page-posts to render a page for a list of posts, part-header to render the header, part-footer to render the footer, part-nav to render the navigation menu.

5.7 The App

5.7.1 The main files

index.js

The index.js contains the code to start the website.
In Desidera, a website results to be a content-driven single-page progressive-web-app.

index.html

The current Web is actually driven by HTML: if you want to run a script in a web browser, you need to load an HTML document that imports the code to run by means the `<script>` tag.
In Desidera, the index.html is used just to import the starting code contained in `index.js`.

```
<!DOCTYPE html>
<html>
<head>
  <title>Desidera</title>
  <meta charset="utf-8">
</head>
<body>
  <script type="module" src="index.js"></script>
</body>
</html>
```

5.7.2 Routing

In Desidera, routing treat routes like any other application data/state.

In traditional routing, a specific route is associated with a route handler that would link your route to a specific view/component.

In Desidera, rather than the router updating the view, the view listens/subscribes to route changes in order to update itself.

On a route change, you only need to re-render a portion of your app. Depending on where you come from, for the same given route, a smaller or larger part of your application view will need to be re-rendered. This is why route handlers are not helpful: routing is not about mapping a route to a component, it is about going from a place to another.

The router just takes navigation instructions and output state updates.

Updating the browser history or listening to URL changes is considered a side-effect, because they are specific to the environment where the website run (the browser).
In Desidera, the router updates the browser URL and translate URL change events to routing instructions.

Tree of routes

Routes are organised in a tree, made of segments and nodes.
At the top will always be an unnamed root node (its name is an empty string).
Each node of the tree (except the root node) is a valid route.
A node of the tree is directly linked to its descendant node.
Links between nodes are called segments.

Route transition

During a transition phase, the router will follow a transition path: it will deactivate some segments and activate some new ones. The intersection node between deactivated and activated segments is the transition node.

For example:

```
├── pages
|    ├── pages.page_1
|    └── pages.page_2
|
└── posts
     ├── posts.post_1
     └── posts.post_2
```

Where:

- the root node is directly linked to `pages` and `posts` nodes.
- `pages` note is directly linked to `pages.page_1` and `pages.page_2`
- `posts` note is directly linked to `posts.post_1` and `posts.post_2`

In a transition from `pages` to `posts.post_1`:

- the root node is the transition node
- the `pages` segment is deactivated
- the `posts` segment is activated
- the `posts.post_1` segment is activated

In a transition from `posts.post_1` to `posts.post_2`:

- the `posts` node is the transition node
- the `posts.post_1` segment is deactivated
- the `posts` segment remains activated
- the `posts.post_2` segment is activated

The router is unaware of your view and you need to bind your view to your router's state updates.

5.7.3 The main process

When the website is open (the `index.html` is fetched), Desidera executes the following steps:

- fetch the `index.json` file
- parse the sitemap contained in the `index.json` file
- for each page (each meta entry), register the relative route
- trigger the event ready
- check if the location hash already contains a path
- load and parse the metadata for the requested page (or for the homepage)
- instantiate the page component for the requested page (using the metadata)
- unmount the currently mounted page component (if any)
    - the page component trigger the event disconnect
- mount the page component in the app container
    - the page component trigger the event connect
- trigger the event open

The app exposes a data object that describe the website. This object contains a content object and a theme object, that mirror the directory structure of, respectively, the content folder and the theme folder.

When a page component is mounted, it takes the control. A mounted page component has access to the app data and the routing data.

5.8 The workflow

5.8.1 Updating a website

To update a website, run just one command:

```
desidera update
```

One of the main advantages to have the content separated from the structure/presentation, is that any update to a file of the theme folder doesn't affect any file of the content folder, and vice versa.

This is very important in a platform where files are distributed and cached over multiple nodes, especially if these files represents a content-centric websites, because content, that is nearly immutable, doesn't need to be rebuilt if its presentation changes.

Running update to a website that has not been modified doesn't produce any effect. Any update is not propagated to the distributed network until the website is not published.

5.8.2 Publishing a website

To publish a website, run just one command:

```
desidera publish
```

The command publish takes the hash of the last published website, and push it in an array of previous versions, then add the current version to the IPFS network, using the IPFS API.

Everywhen the website is updated, Desidera updates the IPNS address to point to the latest version of the website.

A link to a website explicitly refers to a specific version of that website. However, in some cases, it's desirable to have an ever-updated link to the last version of a website. Actually, even updating a single link would trigger a waterfall of updates. For this reason, the website should be published/linked using an IPNS address.

IPNS is a way to add a small amount of mutability to the permanent immutability of IPFS. It allows to store a reference to an IPFS hash under the namespace of the id of the IPFS node that publish the content.

5.8.3 Linking a website

Once a website is published, its pages are ready to be linked by other webpages, or to be pinned by other IPFS nodes.

A resource distributed over IPFS is content-addressed: its address is the cryptographic hash of its content. Since the hash changes if the content changes, every version of the website has a different address. This prevents links to be broken, since at least one node in the IPFS network maintains a copy of the linked website.

The address of a resource is composed by: the hash of the website root folder, followed by the path to the resource (starting from the root folder).

As the website is a SPA, and there is no server, a rendered page address uses the fragment identifier. The address of a rendered page is composed by: the hash of the website root folder, followed by the fragment identifier `#`, followed by the path to the page folder.

CHAPTER 6
PANTAREI
THE RESILIENT FRONT-END WEB FRAMEWORK BASED ON WEB COMPONENTS

In this chapter, a resilient front-end framework, *Pantarei*, is presented.

Pantarei allows you to build interactive Web Components, and componentize your website.

There are two prevailing approaches for how to create components: “abstract the platform” or “embrace the platform”.

Abstract the platform means to avoid HTML, CSS, and DOM as much as possible, and to write everything in JavaScript. Then use all of the features of JavaScript to componentize the website. This is the approach of React[203], the Web UI Framework promoted by Facebook.

Embrace the platform means to use the recently added componentization features of the Web platform itself. This is what the W3C Web Components spec is all about[204]. This is the approach of Polymer[205], the Web UI Framework promoted by Google.

Pantarei embraces the platform. It is based on the HTML5 Web Components standards, in order to ease the process to componentize websites, to design components that can be generally used in the most natural and longeval way for the Web: using Web Standards.

---

[203] Facebook. React.js. (https://goo.gl/pCTL56)
[204] D. Cooney. (July 2014). *Introduction to Web Components*. (https://goo.gl/geUZLf)
[205] Google. Polymer Project. (https://goo.gl/uCwnHt)

6.1 Motivation

There is no task in software engineering today quite as challenging as web development. To tackle the problem of building attractive, interactive user interfaces for the Web, the web development community has evolved a vast ecosystem of web frameworks.

A framework is not just a mere technology, a framework expresses a design philosophy. Developers might like to think that they are in control of their tools, that they bend them to their will, but the truth is that all software is opinionated software.

> S*oftware, like all technologies, is inherently political. Code inevitably reflects the choices, biases, and desires of its creators*.[206]

Choosing a framework is the main decision that strongly influences the overall system design. Web frameworks tend to be overarching in every part of the project, leading to lock-in from day one.

Migrating off from a framework, or sometimes even upgrading a framework, is so challenging, that usually switching to another framework means starting the entire project from scratch. But thinking of not changing, or not upgrading, wouldn't simply reflect the reality: Web application requirements and device and browser capabilities are constantly evolving, and the Web development community is constantly craving the next new thing.

Web frameworks came and go. Here one day, out the next. Hot today, obsolete in a year. There is the risk to pick something it'll be irrelevant long before the life of the project built with it.

To finally solve the "Framework Churn"[207], it's important to understand what a modern front-end Web framework is and why it causes lock-in and subsequent violent churn as another framework shows web developers to better grasp the design needs of the moment.

---

[206] Jamais Cascio. (November 2007). *Openness and the Metaverse Singularity*. (https://goo.gl/vYm8wV)
[207] Max Lynch. (September 2017). *The framework churn problem*. (https://goo.gl/zK1ehr)

Fundamentally, a modern frontend framework provides a component model, e.g Angular, Vue, React, etc., define its own component model, and a way to make components reactive, e.g. implementing Virtual DOM or binding the data model to element directives.

A component model specifies how components work along with tools for templating, component loading and more, how they expect to be created, initialized, rendered, updated, destroyed, how they expect to be customized and stylized, and especially how they interact with each other.

Components built in one framework do not work in another framework, due to the systems the framework provides to make the components run. This makes frameworks obsolete.

Fortunately, a new Web Standard has been defined: the HTML5 Web Components. After being slowly implemented by browser vendors[208], actually, Web Components is supported by all the major browsers.

HTML5 Web Components provide a standardized way to build components for the Web but does not provide any support for dynamic rendering, reactivity, UX patterns, etc.

Enter Panterei. Pantarei is a light layer built on top of the HTML5 Web Components standards, that allow you to build resilient, reactive Web Components, in order to ease the process to extend and customize websites in the most natural and longeval way for the Web: using Web Standards.

---

[208] Alex Rauschmayer. (August 2015). *What happened to Web Components?* (https://goo.gl/6PZspi)

6.2 Principles

Every framework conveys a design philosophy, even if not explicitly. After all, even "there are no rules" is a rule.

Every choice in computing, from data formats to algorithms, has a tradeoff. And even with a great deal of planning, decisions may lead to breaking changes down the road. Allowing systems to evolve and grow is important, rather fundamental. Pantarei paves the way for the resilience: it pursuits the *separation of concerns* in every aspect of the design.

Pantarei makes a clear separation between how the UI is componentized and how the UI is made reactive: it uses components to encapsulate UI units that have their own view and data logic; it uses directives to declare and isolate reactive behavior.

Pantarei makes a clear separation between data logic and presentation: it isolates logic in JavaScript classes, and presentation in HTML templates; it doesn't fill the HTML templates with imperative control flow, nor any logical expression.

Why the name Pantarei

The expression φαντα-ρει, from greek panta-rei, that means "everything flows", either was spoken by Heraclitus or survived as a quotation of his.

> *All entities move and nothing remains still.*
> *Ever-newer waters flow on those who step into the same rivers.*
> *—Heraclitus*

As a Web Framework, "panta-rei" recalls the mutability of the views as well as the flow of the data that make the views reactive.

6.3 Directives

Pantarei makes the view reactive by means directives.

Directives are special attributes attached to an element, that tells the element what to do everywhen the data associated with the element changes.

For example, a directive could tell an element to: update an attribute, update a property, update the inner text, update the style, handler an event, etc.

Basically, a directive is assigned to an element simply as a common attribute.
An element can have more than one directive assigned to it.

In general, in Pantarei a directive is defined as follows:

`bind-`*`directive-key`*`=“`*`value`*`”`

Where:

- `bind-` is a prefix to disambiguate data binding attributes
- *`directive`* is the name of the directive
- *key* is a parameter name used by the directive
- *value* is the name of a property/expression/function assigned to the element that determines the reactiveness of the element

For example:

```
<a id="link" bind-attr-href="url">this is a www link</a>
```

```
let link = window.link
link.url = "http://www.example.com"
assert(link.attributes.href === link.url)
```

6.3.1 Syntax

In Pantarei, a directive is simply identified by the namespace.

In Pantarei, unlike other Web frameworks, the syntax used for directive is compliant to the HTML spec[209].

In Pantarei, unlike other Web frameworks, a directive does not contain any logic instruction nor any boolean expression, but only the name of the data property that could trigger the directive, that could be a property of the data associated to the element, or a method of the element itself, for example used as an event handler.

In Angular, for instance, the official guide suggests to keep simplicity[210]:

> *Although it's possible to write quite complex template expressions, you should avoid them. Confine application and business logic to the component itself, where it will be easier to develop and test.*

In Pantarei, template expressions are not allowed at all.

[209] W3C. *The HTML syntax*. (https://goo.gl/MnJGGg)
[210] Google. Angular.js. *Template Syntax: Simplicity* (https://goo.gl/obKxin)

6.3.2 The Element constructor

To make an element reactive, Pantarei provides the constructor `Element`, that is an extension of the `HTMLElement` native class.

Given an HTML element (to be made reactive), and a data object (to trigger the reactions), the constructor `Element` executes the following procedure to the element and recursively for each child of the element:

- for each attribute of the element:
    - parse the attribute using the directive parsers
    - if the parsed attribute represents a known directives:
        - create the directive using the directive constructor
        - assign the directive to element

Once an element is parsed and bounded to a data object, it reacts to the changes of the bounded data object.

In the following example, everytime the data property count is incremented, the element inner text updates.

```
<div id="el" bind-text="count"></div>
```

```
let element = window.el
let data = { count: 0 }
let app = new Pantarei.Element({ element, data })

function tick () {
  app.data.count++
}

window.setInterval(tick, 1000)
```

6.3.3 Built-in directives

Pantarei comes with a minimal set of built-in directives, that correspond to the basic way an element could be updated:

- `DirectiveAttribute`, to update the element attribute
- `DirectiveProperty`, to update the element property
- `DirectiveClassName`, to update the element class list
- `DirectiveText`, to update the inner text
- `DirectiveRepeat`, to repeat rendering
- `DirectiveToggle`, to toggle rendering
- `DirectiveEvent`, to handle native and custom events

6.3.4 Update an attribute

The directive attribute allows you to bind an element's attribute with a data property.

By default, it is defined as follows:

```
bind-attr-attribute="prop"
```

Where:

- `bind-attr-` is the prefix to identify a directive attribute
- *`attribute`* is the name of the element attribute to be bound
- *`prop`* is the name of the data property to be bound

In practice, the value of the attribute *attribute* is bound to the value of the data property *key*. When the value of the data property *key* changes, the value of the attribute *attribute* changes.

In the following example, the element attribute title is bound to the data property message.

```
<a id="element" bind-attr-title="message">link</a>
```

```
let element = document.element
let data = {
  message: "Hello Pantarei!"
}
let app = new Pantarei.Element({ element, data })
```

6.3.5 Update a property

The directive property allows you to bind an element's property with a data property.

By default, it's defined as follows:

`bind-prop-`*`property`*`="`*`prop`*`"`

Where:

- `bind-prop-` is the prefix to identify a directive property
- *`property`* is the name of the element property to be bound
- *`prop`* is the name of the data property to be bound

In the following example, the element property disabled is bound to the data property test.

```
<button id="el" bind-prop-disabled="test">button</button>
```

```
let element = window.el
let data = {
  test: true
}
let app = new Pantarei.Element({ element, data })
```

HTML Attributes vs. DOM properties

Attributes are defined by HTML. Properties are defined by the DOM.

- A few HTML attributes have 1:1 mapping to properties, for example id.
- Some HTML attributes don't have corresponding properties, for example colspan.
- Some DOM properties don't have corresponding attributes, for example textContent.
- Many HTML attributes appear to map to properties in particular ways.

The HTML attribute value specifies the initial value. The DOM value property specifies the current value.

For example, the following HTML element creates a corresponding DOM node with a value property initialized to "Alice".

```
<input type="text" value="Alice">
```

Where:

- if the user enters "Bob" into the input field, the DOM element value property becomes "Bob", but the HTML value attribute remains unchanged (to "Alice").

The HTML attribute and the DOM property are not the same thing, even when they have the same name.

For example, the HTML button disabled attribute initializes the button's disabled property to true just with its presence alone. Adding and removing the disabled attribute disables and enables the button. Changing the value of the attribute is irrelevant.

```
<button>Enabled</button>
<button disabled>Disabled</button>
<button disabled="false">Still Disabled</button>.
```

6.3.6 Update the class list

The directive class allows you to toggle an element's classname.

By default, It is defined as follows:

```
bind-class-classname="prop"
```

Where:

- class. is the prefix to identify a directive class
- classname is the classname of the element classlist to be bound
- value is the name of the data property used to toggle the classname

In practice, if toggle is true, the class classname is added to the classlist, otherwise, it's removed (if present).

For example:

```
<style>
  #el {
    visibility: hidden;
  }
  #el.visible {
    visibility: visible;
  }
</style>

<div id="el" bind-class-visible="show">Hello Web!</div>
```

```
let element = window.el
let data = {
  show: true
}
let app = new Pantarei.Element({ element, data })

function toggle () {
  app.data.show = !app.data.show
}

window.setInterval(toggle, 1000)
```

6.3.7 Update the inner text

The directive text allows you to update the text that is inner (or before, or after) an element.

By default, it is defined as follows:

```
bind-text="prop"
```

Where:

- `bind-text` is the name of the directive Text
- *prop* is the name of the data property to be bound For example:

```
<div id="el" bind-text="message"></div>
```

```
let element = window.el
let data = { message: "Hello Web!" }
let app = new Pantarei.Element({ element, data })
```

6.3.8 Repeat rendering

The directive repeat allows you to repeat the elements according to a list of items.
The element to be repeated should be contained in a template element.

By default, it is defined as follows:

```
for-items="value"
```

Where:

- `for-items` identifies a directive repeat
- *`value`* is the name of the data property (that must be an array) to be bound

For example:

```
<ul id="el">
  <template for-items="items" for-item="item">
    <li bind-text="item"></li>
  </template>
</ul>
```

```
let element = window.el
let data = {
  items: ["a", "b", "c"]
}
let app = new Pantarei.Element({ element, data })
```

Nested repeat

Nested repeat is also allowed. For example:

```
<div id="el">
  <template for-items="table" for-item="row">
    <template for-items="row" for-item="col">
      <div bind-text="col"></div>
    </template>
  </template>
</div>
```

```
const element = window.el
const data = {
  table: [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
  ]
}
const app = new Pantarei.Element({ element, data })
```

The repeated element can access their relative item through the object specified in the repeat.item attribute. To change the name of this accessor, the directive repeat assigned to a template element, check for the presence of the repeat.item attribute, and if present, use its value to rename the item, otherwise use the name item.
Renaming the item is indispensable to avoid conflicts in nested repeat. In the previous example, the iterated items are renamed row and col.

6.3.9 Conditional rendering

The directive toggle allows you to toggle an element.
The element to be toggled should be contained in a template element.

By default, it is defined as follows:

```
if-boolean="value"
```

Where:

- `if-` identifies a directive toggle
- *`boolean`* is the boolean value to test, that must be true or false
- *`value`* is the name of the data property to be bound

If the value of value is equals to the value expressed by boolean, then the element will be shown, otherwise it will be hidden.

For example:

```
<div id="el">
 <p if-true="truly">visible</p>
 <p if-true="falsy">hidden</p>
 <p if-false="truly">hidden</p>
 <p if-false="falsy">visible</p>
</div>
```

```
let element = window.el
let data = {
  truly: true,
  falsy: false
}
let app = new Pantarei.Element({ element, data })
```

6.3.10 Handling Events

The event directive allows you to bind handlers to events (let them be custom or native). The type name of the directive is ev. Event handling is based on event delegation: registering an event to a repeated element results in just one listener.

By default, it is defined as follows:

`on-`*`event`*`="`*`handler`*`"`

Where:

- `on-` identifies a directive event
- *`event`* is the event name
- *`handler`* is the name of the method that handles the event

For example:

```
<ul id="el">
  <template repeat-items="numbers">
    <li on-click="on_click" bind-text="item"></li>
  </template>
</ul>
```

```
let element = window.el
let data = {
  items: [1, 2, 3]
}
let handlers = {
  on_click (event) {
    alert('Clicked number ${event.target.innerText}')
  }
}
let app = new Pantarei.Element({ element, data, ...handlers })
```

6.4 Custom directives

Pantarei allows you to create new directives, as well as customize existing ones.

6.4.1 Create a custom directive

Directives extends from the parent `Directive` class.
To create a new directive you must extend the Directive class.

The `DirectiveParser` class exposes the following interface, that must be overridden:

- `static match(attr)` to check if the attribute `attr` is a directive
- `static parse(el, attr)` to parse the directive and assign it to `el`

The `Directive` class exposes the following interface, that must be overridden:

- `constructor(el)` to bind this directive to the element `el`
- `run(data)` to run the directive on the binded element (`el`) using `data`

Install a new directive

To install a custom directive (e.g., `MyCustomDirective`), just add it to the array of directives of the `Pantarei.Element` class.

```
Pantarei.Element.directives.push(MyCustomDirective)
```

If a directive is added after the element has been initialized, it will not have any effect.

6.4.2 Extend built-in or custom directives

The way to be resilient pass through the ability to customize what is already provided.

Customizing a built-in directive could be useful to change the default behaviour or even the syntax used by the framework to identify the directive.

For example, In the following example, the definition of the directive attribute is modified to match the following syntax (similar to the syntax used by Angular and Vue):

```
[attribute]="prop"
```

Where:

- *attribute* is the name of the element attribute to be bound
- *prop* is the name of the data property to be bound

```
class AttributeDirectiveParser extends DirectiveParser {
  static match (attribute) {
    return (/\[\w+\]/gi).test(attribute.name)
  }

  static parse (element, attribute) {
    const regexp = /\[\(w+)\]/gi
    const matches = regexp.exec(attribute.name)
    const key = matches[1]
    const value = attribute.value
    const directive = new AttributeDirective({ element, key, value })
    return directive
  }
}
```

The customized directive attribute can be used as follow:

```
<a id="el" [title]="message">link</a>
```

```
let element = document.element
let data = { message: "Hello Web!" }
let app = new Pantarei.Element({ element, data })
```

6.4.3 Directives mapping

One of the most powerful aspect of directives in Pantarei is that they could be distributed over a DOM tree (an element and its children) dynamically, in a so called “transparent” way, using the `directives` mapping.

```
{
  "element_selector": {
    "directive_key": "directive_value",
    …
}
```

The directives object option acts in a similar way of a stylesheet.
We could say, a directive mapping is for reactivity as a stylesheet is for styling.

In this way, the presentation is completely separated from the data logic: the HTML is totally clean and independent from any additional syntax.

For example:

```
<div id="el">
  <button id="button"></button>
</div>
```

```
{
  "button": {
    "on-click": "on_click",
    "on-over": "on_over",
    "bind-text": "message"
  }
}
```

6.5 Components

Pantarei allows you to build components to componentize the app. Components help extend basic HTML elements to encapsulate reusable code.

Pantarei Components provide syntactic sugar to Custom Elements definition, using HTML Template and Shadow DOM.

6.5.1 Component definition

Using HTMLElement:

```
let content = `
<style>/* scoped style */</style>
<div>Shadow DOM</div>
`

class MyElement extends HTMLElement {

  constructor () {
    super()
    const shadow = this.attachShadow({mode: 'open'})
    const template = document.createElement('template')
    template.innerHTML = content
  }

  connectedCallback () {
    console.log('connected')
  }

  disconnectedCallback () {
    console.log('disconnected')
  }

  adoptedCallback () {
    console.log('adopted')
  }

  attributeChangedCallback () {
    console.log('attribute changed')
  }

}

window.customElements.define('my-element', MyElement)
```

Using Pantarei.Element:

In `my-element/template.html`:

```
<div>Shadow DOM</div>
```

In `my-element/style.css`:

```
/* scoped style */
```

In `my-element/index.js`:

```
class MyElement extends Pantarei.Element {
  static is = 'my-element'

  static style_urls = ['./style.css']

  static template_url = './template.html'

  constructor () {}

  ready () {
    console.log('ready')
  }

  connected () {
    console.log('connected')
  }

  disconnected () {
    console.log('disconnected')
  }

  adopted () {
    console.log('adopted')
  }

}
```

6.5.2 Component properties

Every component instance has its own isolated scope. This means you can not (and should not) directly reference parent data in a child component's template.

Data can be passed down to child components using the `props` option. A property defined in `props` is a custom property for passing information from parent components. A child component needs to explicitly declare the properties it expects to receive using the `props` option.

For example:

```
class MyComponent extends Pantarei.Component {

  static is = "my-component"

  static props = ["test"]

}
```

```
const my_component = document.createElement("my-component")
my_component.test = "This will trigger re-rendering"
```

Properties validation

It is possible to specify requirements for the properties of a component.
This is especially useful for the component documentation, maintenance and reuse.

The type can be one of the following native constructors: `String`, `Number`, `Boolean`, `Date`, `Function`, `Object`, `Array`, `Symbol`. In addition, type can also be a custom constructor function and the assertion will be made with an `instanceof` check.
When property validation fails, a console warning will be produced.

For example:

```
class BlogPost extends Pantarei.Component {

  static is = 'blog-post'

  static props = {
    "title": {
      type: String
    },
    "author": {
      type: String
    },
    "is_draft": {
      type: Boolean,
      value: true
    },
    "excerpt": {
      type: String
    },
    "content": {
      type: String
    },
    "published_at": {
      type: Date
    }
  }

}
```

6.5.3 Composition

Components are meant to be used together, most commonly in parent-child relationships. Component A may use component B in its own template. They inevitably need to communicate to one another: the parent may need to pass data down to the child, and the child may need to inform the parent of something that happened in the child.

However, it is also very important to keep the parent and the child as decoupled as possible via a clearly-defined interface. This ensures each component's code can be written and reasoned about in relative isolation, thus making them more maintainable and potentially easier to reuse.

In Pantarei, the parent-child component relationship can be summarized as "props go down, events bubble up". The parent passes data down to the child via props, and the child sends messages to the parent via events.

6.5.4 Distribution

When using components, it is often desired to compose them.
Pantarei rely on Web Components, so it can leverage on the special `<slot>` element to serve as distribution outlets for the original content.

For example:

```
<app-container>
  <part-header slot="header"></part-header>
  <div id="content" slot="main"></div>
  <part-footer slot="footer"></part-footer>
</app-container>
```

There are two things to note here.
The `<app-container>` component does not know what content it will receive. It is decided by the component using `<app-container>`.
The `<app-container>` component very likely has its own template. To make the composition work, we need a way to interweave the parent "content" and the component's own template. This is a process called content distribution.

6.6 Designing a website

6.6.1 Content types

In `types/post/data`:

```
{
  "name": "post",
  "plural": "posts",
  "type": "object",
  "properties": [
    { "name": "title", "type": "text", "required": true },
    { "name": "excerpt", "type": "string" },
    { "name": "content", "type": "string" },
    { "name": "published_at", "type": "datetime" },
    { "name": "is_draft", "type": "boolean" }
  ],
  "controls": [
    { "property": "title", "widget": "single_line" },
    { "property": "excerpt", "widget": "multiple_line" },
    { "property": "content", "widget": "multiple_line" },
    { "property": "published_at", "widget": "date_picker" },
    { "property": "is_draft", "widget": "boolean" }
  ]
}
```

In `types/image/data`:

```
{
  "name": "image",
  "plural": "images",
  "type": "object",
  "properties": [
    { "name": "name", "type": "string" },
    { "name": "description", "type": "string" },
  ],
  "controls": [
    { "property": "name", "widget": "string" },
    { "property": "description", "widget": "string" }
  ]
}
```

In `types/author/data`:

```
{
  "name": "author",
  "plural": "authors",
  "type": "object",
  "properties": [
    { "name": "full_name", "type": "string" },
    { "name": "picture", "type": "image" }
  ],
  "controls": [
    { "property": "full_name", "widget": "string" },
    { "property": "picture", "widget": "image" }
  ]
}
```

6.6.2 Content models

In `models/posts/post_1/data`:

```
{
  "title": "My first post",
  "draft": false
}
```

In `models/posts/post_1/content`:

```
# This is my first post.

The following is a list:

- one
- two
- three
```

In `models/posts/post_2/data`:

```
{
  "title": "My first post",
  "draft": false
}
```

In `models/posts/post_1/data`:

```
# This is my first post.

The following is a list:

- one
- two
- three
```

The directory structure is:

```
models
├── pages
|    ├── page_1
|    |    ├── data
|    |    ├── content
|    |    └── cover
|    |
|    ├── page_2
|   ...   ├── data
|         ├── content
|         ├── cover
|         ├── page_2_1
|              ├── data
|              ├── content
|              └── cover
|
├── posts
|    ├── post_1
|    |    ├── data
|    |    ├── content
|    |    └── cover
|    ├── post_2
|   ...   ├── data
|         ├── content
|         └── cover
|
├── authors
|    ├ author_1
|    … ├── data
|        └── picture
…
```

Routes

In `routes.json`:

```
[
  {
    "name": "home",
    "path": "/",
    "component": "page-home"
  },
  {
    "name": "pages",
    "path": "/pages",
    "component": "page-list"
  },
  {
    "name": "page",
    "path": "/pages/:path*",
    "component": "page-single"
  }
  {
    "name": "posts",
    "path": "/posts",
    "component": "page-posts"
  },
  {
    "name": "post",
    "path": "/posts/:post_id",
    "component": "page-post"
  },
  {
    "name": "tags",
    "path": "/tags",
    "component": "page-tags"
  },
  {
    "name": "tag",
    "path": "/tags/:tag_id",
    "component": "page-tag"
  }
]
```

6.6.3 Theme

```
theme
├── config.json
├── index.js
├── components
...   ├── page-home
      |    ├── index.js
      |    ├── template.html
      |    └── style.css
      |
      ├── page-post
      ├── page-posts
      ├── page-category
      ├── page-categories
      ├── page-tag
      ├── page-tags
      ├── page-search
      ...
      ├── part-header
      ├── part-footer
      ├── part-menu
      ├── part-breadcrumbs
      ├── part-cover
      ├── part-content
      ...
```

Page Posts

In /theme/components/page-posts/template.html:

```
<div id="page">
  <h1 id="title"></h1>
  <div id="posts">
    <div id="post">
      <h2 id="title"></h2>
      <div id="excerpt"></div>
      <div id="published_date"></div>
    </div>
  </div>
</div>
```

In /theme/components/page-posts/directives.json:

```
{
  "#title": {
    "bind-text": "data.title"
  },
  "#posts": {
    "repeat-items": "data.posts",
    "repeat-item": "post"
  },
  "#posts #post #title": {
    "bind-text": "post.title"
  },
  "#posts #post #excerpt": {
    "bind-text": "post.excerpt"
  },
  "#posts #post #published_date": {
    "bind-text": "post.published_date"
  }
}
```

In /theme/components/page-posts/index.js:

```
export class PagePost extends Page {

  connected (context) {
    this.action('fetch_posts')
  }

}
```

Page Post

In `/theme/components/page-post/template.html`:

```
<div id="post">
  <h1 id="title"></h1>
  <img id="cover" />
  <div id="content"></div>
</div>
```

In `/theme/components/page-post/directives.json`:

```
{
  "#title": {
    "bind-text": "data.title"
  },
  "#cover": {
    "bind-attr-src": "data.cover"
  },
  "#content": {
    "bind-text-md": "data.content"
  }
}
```

In `/theme/components/page-post/index.js`:

```
export class PagePost extends Page {

  connected (context) {
    let post_id = context.params.post_id
    this.action('fetch_post', { post_id, resolve: { author: true } })
  }

}
```

Page Tags

In `/theme/component/page-tags/template.html`:

```
<div id="page">
  <h1 id="title"></h1>
  <div id="tags">
    <div id="tag">
      <h2 id="title"></h2>
    </div>
  </div>
</div>
```

In `/theme/component/page-tags/directives.js`:

```
{
  "#title": {
    "bind-text": "data.title"
  },
  "#tags": {
    "repeat-items": "data.tags",
    "repeat-item": "tag"
  },
  "#tag #title": {
    "bind-text": "tag.title",
  }
}
```

In `/theme/components/page-tags/index.js`:

```
export class PageTags extends Page {

  connected (context) {
    this.action('fetch_tags')
  }

}
```

Page Tag

In `/theme/component/page-tag/template.html`:

```
<div id="page">
  <h1 id="title"></h1>
  <div id="posts">
    <div id="post">
      <h2 id="title"></h2>
      <div id="excerpt"></div>
      <div id="published_date"></div>
    </div>
  </div>
</div>
```

In `/theme/component/page-tag/directives.json`:

```
{
  "#title": {
    "bind-text": "data.title"
  },
  "#posts": {
    "repeat-items": "data.posts",
    "repeat-item": "post"
  },
  "#post #title": {
    "bind-text": "post.title",
  },
  "#post #excerpt": {
    "bind-text": "post.excerpt"
  },
  "#post #published_date": {
    "bind-text": "post.published_date"
  }
}
```

In `/theme/components/page-posts/index.js`:

```
export class PageTag extends Page {

  connected (context) {
    let tag_id = context.params.tag_id
    this.action('fetch_posts', { tag_id })
  }

}
```

Store

In `/theme/index.js`:

```
export default class extends Pantarei.Store {

  get initial_state () {
    return {
      post: undefined,
      posts: []
    }
  }

  async update_post ({ post_id }) {
    let post = await this.fetch_post({ post_id })
    this.update({ post })
  }

  async fetch_post ({ post_id, resolve }) {
    let post = await ipws.get(`/content/posts/${post_id}`)

    if (resolve.author) {
      let author = await ipws.get(`/content/authors/${post.author}`)
      post.author = autor
    }

    return post
  }

  async fetch_posts ({ limit }) {
    let post_ids = await ipws.get(`/content/posts`)
    let posts = []
    for (post_id of post_ids) {
      let post = await fetch_post({ post_id })
      posts.push(post)
    }
    return posts
  }
}
```

CHAPTER 7
CONCLUSIONS

7.1 The state of the art

There are many ways to build a website:

- Content Management Systems (CMSs)
- Headless CMSs
- Static Site Generators
- Static Progressive Web App Generators

CMS like Wordpress give you an online text editor to create content. They let you customize the look and feel through choosing themes and plugins, or writing custom PHP or Javascript code. Content is saved in a database, and it is retrieved and sent to users when they visit the website. Depending on your requirements you can self-host your website, or use an official hosting provider.

Headless CMSs like Contentful require you to build the front-end side, using a front-end Web framework.

Traditional static site generators like Hugo let you put text or markdown in a specific directory in a version-controlled codebase. They then build a specific kind of site, usually a blog, as HTML files from the content you've added. These files can be cached and served from a CDN.

Progressive website generators like Gatsby let you build progressive web apps, with the advantages of the static websites.

7.2 Pantarei

The primary purpose of Pantarei is to create a resilient Web Framework, to build a reliable system of Web Components to build websites with.

Learning Curve

React, requires learning JSX, as well as CSS-in-JS methodologies for styling components.

In Angular, the API surface of the framework is huge and as a user you will need to familiarize yourself with a lot more concepts before getting productive. The complexity of Angular is largely due to its design goal of targeting only large, complex applications - but that does make the framework a lot more difficult for less-experienced developers to pick up.

In Pantarei, you can start building non-trivial applications, having just familiarity with HTML, CSS and plain JavaScript. It exposes just two basic concepts: directives and components.

JavaScript idioms/extensions

React requires using JSX. Using React without JSX can be challenging.

Angular requires using TypeScript. Using Angular without TypeScript can be challenging.
TypeScript has its benefits - static type checking can be very useful for large-scale apps, and can be a big productivity boost for developers with backgrounds in Java and C#. However, in many smaller-scale use cases, introducing a type system may result in more overhead than productivity gain.

Pantarei requires just plain JavaScript.

Javascript has become more powerful as a language in the last several years, making it easier to write code, and making it useless to use alternative compiled-to-javascript languages.
For instance, the last version of the Standard introduced:

- language features such as modules and classes
- syntactic constructs for asynchronous programming such as async/await
- data structures/types such as maps, sets, weakmap and weakset
- syntactic sugar such as arrow functions, destructuring, template strings

Component ecosystem

React has several sets of out-of-the-box component libraries, as well as curated sets.

Pantarei is bases on HTML5 Web Components. There are thousands of user defined Web Components that can already be used[211].

Optimizations

In React, when a component's state changes, it triggers the re-render of the entire component sub-tree, starting at that component as root. To avoid unnecessary re-renders of child components, you need to programmatically specify the condition in which the component should update, whenever you can.

In Angular, when there are a lot of watchers, rendering becomes slow, because every time anything in the scope changes, all these watchers need to be re-evaluated again. Also, the digest cycle may have to run multiple times to "stabilize" if some watcher triggers another update. In some situations, there is no way to optimize a scope with many watchers.

In Pantarei, as well as Vue, component dependencies are automatically tracked during its render, so the system knows precisely which components actually need to re-render when state changes. This removes the need for a whole class of performance optimizations from the developer's plate, and allows them to focus more on building the app itself as it scales.

[211] WebComponents.org. *List of Web Components*. (https://goo.gl/QPJSY6)

7.3 Desidera

The primary purpose of Desidera is to enable a truly distributed Web, that is: provide an approach to design, build and deploy websites over a content-addressed space delivered by a peer-to-peer network.

7.3.1 Content Delivery

Delivery over a peer-to-peer network

Desidera websites are delivered over the InterPlanetary File System.
Other systems do not support any peer-to-peer network.

Desidera websites result to be:

- censorship-resistant: the content is distributed over a peer-to-peer network; as long as at least one node hosts the website, the website will be reachable.
- permanent: every update generates a new version of the website; previous versions are still linkables.

Delivery over a CDN

Wordpress supports this through customization[212] and plugins[213].
Hugo and Gatsby are built for this.
Desidera websites can be delivery over a CDN as well as over IPFS.

7.3.2 Dev Op Experience

Serverless

Serverless means not having to worry about security and framework upgrades.
WordPress requires a server to generate pages.
Desidera, as well as Hugo and Gatsby, are serverless.

Hosted

Wordpress comes with built-in hosting.
Gatsby and other static site generators can be plugged into static hosts.
Desidera is hosted by the peers of a distributed peer-to-peer network.

---

[212] WordPress. *How to serve static content from a cookieless domain*. (https://goo.gl/MiFGWx)
[213] WordPress. *CDN Plugin* (https://goo.gl/gEWgvT)

7.3.3 User Experience

Offline access

Offline access via Service Workers is one of the core principles of Progressive Web Apps.
Wordpress require maintaining dual PHP and JavaScript templates.
Gatsby supports this out of the box.
Desidera is offline-ready too.

Prefetch linked pages

When a page loads, the content needed to load the next link you click will be loaded in the background while you browse the page.
Wordpress require maintaining dual PHP and JavaScript templates.
Desidera, as well as Gatsby, supports this out of the box.

Page caching

Fingerprinting static resources that aren't expected to change lets browsers serve content locally when a user visits a page they've already been to, as opposed to making an extra network call.
Wordpress allows this via plugins.
Desidera, as well as Gatsby support this out of the box.

7.3.4 Front-end Developer Experience

Componentization

Component systems allow developers to plug-n-play either external 3rd party components or internal components from a shared codebase or component library.
WordPress support this through PHP.
Hugo does not support this.
Gatsby supports this through React.
Desidera support this through Pantarei, and HTML5 Web Components.

Declarative component

When components are behaving oddly, you can inspect their state and compare expected state of each element in a hierarchy to the actual state, enabling faster debug cycles compared to alternate frameworks.
Desidera support this through Pantarei, which allows a hierarchical UI construction by declaratively passing properties down child trees.

Unidirectional data flows

One-directional data flows are essential to building complex frontend components by removing complex cross-dependencies present in alternate data flow approaches, such as two-way bindings.
Desidera support this through Pantarei, which uses a one-directional data binding system for its components.

7.4 Future works

The future works goes in two parallel directions:

- improve what already has been defined and designed: *Pantarei* and *Desidera*;
- define and design new tools: *Enquire*.

Pantarei

The core of Pantarei can be extended in two parallel directions:

- create new directives;
- create new components.

For both the directions, creating the developer documentation.

Desidera

The Desidera platform can be extended in two parallel directions:

- create new themes (using Pantarei Components);
- create new plugins (using Node.js modules).

Again, for both the directions, creating the developer documentation.

The User Experience could be improved by:

- create new components;
- create new themes.

The Content-Designer Experience could be improved by the followings:

- create a wysiwyg editor (for Markdown content);
- create a real-time (using IPFS PubSub and HTML5 WebRTC[214]) collaborative (using CRDT[215] - Convergent Replicated Data Types) wysiwyg editor.

The Theme-Designer Experience could be improved by the following:

- improve the default theme;
- create a Visual Composer to build themes.

---

[214] W3C (June 2018). *WebRTC 1.0: Real-time Communication Between Browsers*. (https://goo.gl/HG5uKZ)

[215] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski (January 2011). *A comprehensive study of Convergent and Commutative Replicated Data Types*. (https://goo.gl/7jdtVQ)

7.4.1 Enquire - the Search Engine for the Distributed Web

Move to decentralized systems is not so straightforward. This is especially true for search engines. Creating distributed systems that are nonetheless capable of scaling so that they can index most of the Web is challenging.

Despite the challenge, distributed search engines do already exist, albeit in a fairly rudimentary state, such as YaCy. Far from being a valid alternative to Google as of now, YaCy offers a credible alternative model.

YaCy

YaCy is a free search engine that anyone can use to build a search portal for their intranet or to help search the public internet. When contributing to the world-wide peer network, the scale of YaCy is limited only by the number of users in the world and can index billions of web pages. It is fully decentralized: all users of the search engine network are equal; the network does not store user search requests and it is not possible for anyone to censor the content of the shared index. Currently it has about 1.4 billion documents in its index and more than 600 peer operators contribute each month. About 130,000 search queries are performed with this network each day[216].

Some search engines promise privacy, and while they look like real search engines, they are just proxies. Their results don't come from their own index, but from the big incumbents (Google, Bing, Yahoo) instead: the query is forwarded to the incumbent, and the results from incumbent are relayed back to the user (even Bing uses Google[217]).

---

[216] YaCy. *Web Search By The People for the People*. (https://goo.gl/h18Zrs)
[217] Google Official Blog (February 2011). Microsoft's Bing uses Google search results—and denies it. (https://goo.gl/b3r21W)

7.4.2 Enquire

Enquire is the Distributed Search Engine for the Desidera/IPFS network.

In essence, since Desidera websites are already distributed, nodes of the network can also join creating and sharing search results, using the documents they store. Peers communicate each other using the IPFS real-time messaging layer PubSub[218].
In Desidera, content has a simplified syntax (using the human-readable markdown) which ease the text mining process. Furthermore, content documents are paired with well defined content structures (stored in separated metadata documents, i.e. `metadata.json`, and described in another separated document schema, i.e. `schema.json`) which ease the indexing process.

Actually, designing of *Enquire* is in progress.

Why the name

The name is a tribute to Tim Berners-Lee and its first attempt to create a distributed hypertext "*Enquire*" that formed the conceptual basis for the future development of the World Wide Web[219]. (Without meaning to, *Enquire* is also the anagram of *Enrique*).

[218] Jeromy Johnson (May 2017). *Take a look at pubsub on IPFS*. (https://goo.gl/v1AzEc)
[219] W3C. Tim Berners-Lee longer biography. (https://goo.gl/uQs5pk)

The future

On July 10, 2012, in one of his last public interviews[220], Aaron Swartz said:

> *There are two polarizing perspectives.*
> *"Everything is great": the internet has created all this freedom, and liberty, and everything is going to be fantastic.*
> *"Everything is terrible": the internet has created only tools for cracking down, and spying, and, you know, controlling what you said.*
> *And the truth is that both are true. The internet has been both. And both are kind of amazing and astonishing.*
> *And which won in the long run, is up to us.*

The future is up to us.

The work described in this thesis is a little step in the direction of "*unlock the Web open*"[221], to have a more safer, more open, World Wide Web, as it was originally intended[222]:

> *for everyone.*

[220] Anonymous. (September 2014). *The Story of Aaron Swartz Full Documentary*. (https://goo.gl/sh1MEr)
[221] B. Kahle. (August 2015). *Locking the Web Open: A Call for a Decentralized Web*. (https://goo.gl/fDUmgP)
[222] J. Keith. (November 2013). *This is for everyone*. (https://goo. gl/WofL58)