Webdevelopment

Chapter 1

A Bird's Eye View of the Web

Vint Cerf invented the internet while *Tim Berners-Lee* invented the web.

1.1 Dreams and visions

Ted Nelson coined the term hypertext. His showpiece is his *Xanadu* project.

Douglas Engelbart gave the *Mother of All Demos* in 1968.

1.2 The Web and universality

1.2.1 Origins and aims

In 1989, Berners-Lee convinced CERN to let him work on the Web with his *Information Management: A Proposal*. A year later, the first webpages came online.

CERN decided to make the Web available royalty-free in 1993.

Before the Web, the world was heterogenous. It was difficult to target soft- and hardware.

In contrast, the Web strives to be *universal*. Anyone is free to build for the Web. Although, app stores are a counterexample to the freedom to develop for the Web.

Web technologies are standardized by the World Wide Web Consortium (W3C)

The Web brings *permissionless innovation*. You don't need anyone's permission to launch a new idea.

1.2.2 Technical foundations

The Web is an application that runs on top of the Internet. "The *Internet* is to the *Web* what the *telephone network* is to *fax*."

Examples of applications which are on the Internet, but not on the Web: Skype, BitTorrent, e-mail, VPN, ...

The Web as a decentralized hypertext system consists of simplistic one-way links. Text and hyperlinks are contained in an HTML document. These are *not shareable*. Styling, media and scripts are stored outside an HTML document. These *are* shareable.

These realizations contrast Ted Nelson's views.

"Individual links are allowed to break so that the entire Web does not." – Tim Berners-Lee

1.3 Opportunities and challenges

1.3.1 The Web for machines

The Web's architecture makes it hard to locate specific content. *Crawlers* sprung into life to index content (in a centralized manner).

RSS feeds contain the title and summary of the last entries on a website. Here, the user is in control of what they are fed and not the other way around.

The Web as it is, is designed for humans. It is hard for machines to understand the Web. Web APIs expose functionality to automated clients.

The Semantic Web is a layer on top of the existing Web. We would like that the Semantic Web integrates into the existing Web.

In the early days of the Semantic Web, noone built applications, because there was no data. Noone published data, because there were no applications.

Berners-Lee proposed *Linked Data*: principles to follow to publish data semantically. Let's just get data out and apps will follow.

1.3.2 Threats to universality

Native apps essentially undo all progress on device-independence. In contrast, we need but only a generic browser to browse the Web.

Certain browser vendors provide software to access to the web. A few search engine companies make or break websites.

Centralized platforms control data and identities. Ironically, permissionless innovation enables platforms (like Facebook) that prevent it.

Chapter 2

Web Architecture & Technologies

The Web consists of 3 separate but connected inventions. A URL uniquely identifies a resource. HTTP allows one to retrieve the resource identified by a URL. An HTML document may represent a resource or link to other resources through their URL.

2.1 Core Web Standards

2.1.1 URL

A Web URL uniquely *identifies* and *locates* a resource anywhere in the universe.

A string is a unique *identifier* if at most one entity corresponds to it, e.g. a national number uniquely identifies a person, but does not allow locating them.

A string is a unique *locator* if at most one location corresponds to it, e.g. a street address uniquely identifies a location, but does not allow uniquely identifying a person.

URL became part of a family of technologies related to identification. An example of a URL is *mailto:ruben.verborgh@ugent.be*, where Ruben's mailbox is uniquely identified and located. An example of a URN (*Uniform Resource Name*) is an ISBN, which identifies a book in a location-independent manner. URIs (*Uniform Resource Identifiers*) are a union of URLs and URNs. IRI (*Internationalized Resource Identifier*) extends URI with non-ASCII characters.

The uniform structure of an HTTP URL includes the following

```
http://<host>/<path>?<query>#<fragment>
```

where **host** identifies the target machine, **path** identifies the resource within the machine, **query** optionally refines the resource and **fragment** optionally identifies a fragment of the resource.

An HTTP URL contains instructions to obtain a representation of a resource.

1. Using DNS, the client looks up the host's IP address.
2. The client then requests `/<path>?<query>`.
3. The client finds `#fragment`.

2.1.2 HTTP

HTTP is a protocol that standardized client-server communication. HTTP standardizes how clients *request* resource representations from servers. HTTP standardizes how servers reply with a *response* that may contain a representation.

A client sends a request of the following form.

```
GET <path> HTTP/<version>
<headers>
<body>
```

An HTTP method is *safe* if it is read-only. Examples are GET or HEAD.

An HTTP method is *idempotent* if alterations don't alter the outcome. Examples are any safe method, PUT or DELETE.

A client sends a `Host` header with the hostname, so one server may host multiple websites. There is no one-to-one mapping from hostnames to IPs, one website may be hosted on multiple servers. As well as one server may host multiple websites.

A server generates a response of the following form.

```
HTTP/<version> <status> <message>
<headers>
<body>
```

HTTP status codes in a nutshell.

- 1xx: Hold on.
- 2xx: Here you go.
- 3xx: Go away.
- 4xx: You messed up.
- 5xx: I messed up.

2.1.3 HTML

HTML is markup language that captures the structure of documents.

2.2 Web Architecture

2.2.1 Clients

Interactive graphical browsers, applications, crawlers, embedded devices and sensors (Web of Things). Clients need to support TCP/IP, DNS, HTTP and one or more representation formats like HTML.

Browsers render HTML elements as controls, they generally support media while standards (mostly) govern consistency.

Web applications perform HTTP requests using the browser scripting functionality. The server typically returns XML or JSON which is then used by the script. Alternatively, a server may respond with HTML, which may be used to update parts of a page.

Crawlers process and/or index webpages and follow links to other webpages. They may analyze some structured annotations. For example, Google's Rich Snippets or Facebook's Open Graph.

2.2.2 Servers

The HTTP protocol does not attach meaning to URL paths. There exist *file servers* for static files, *application servers* for editable content (CMS) or proxies which delegate requests to other Web servers.

A static file server maps HTTP URLs to internal file URLs.

An application server uses server-side code to generate pages on demand.

The request is (typically) parsed by an application framework, which exposes the URL, method and headers. Implementors may react to specific URLs or patterns, typically generating templated responses.

2.2.3 Intermediaries

There may be many intermediaries between a client and server. Intermediaries may play different roles, such as **caching** to improve performance and availability, **security** to handle authorization and authentication, **routing** to redirect towards the appropriate server, **load balancing** to distribute load over servers or **anonymizing** to bypass identification or logging.

HTTP can be transparent because of its standardized uniform interface. Caching is possible because of headers like `Cache-Control` or ETag.

The standardized method semantics are crucial to making caching work. Because GET is a safe method, repeated GET requests may be served from cache.

If POST or PUT are used on a resource, a subsequent GET must not be read from cache. Repeated PUTs may be ignored, however.

Forward proxies are in the network of the client. Typically used for caching, but possibly for security or anonymity.

Reverse proxies are in the network of the server. Also typically used for caching, also for routing or abstracting remote architecture.

A device may listen on only one TCP port 80, while a reverse proxy may allow it to run many servers. Each server may be run on internal ports (3000, 8000, 8888, ...) instead of port 80. A reverse proxy (NGINX, Apache, ...) dispatches the request appropriately based on `Host` header and/or path.

2.3 Beyond the Core

2.3.1 HTTPS

HTTP nodes send plaintext over TCP, which means intermediaries may read it. The privacy of a client's requests is not guaranteed. The privacy of a server's response is not guaranteed. The integrity of a server's response is not guaranteed.

Simply use HTTP over TLS as you would normally use HTTP over TCP.

Setting up HTTPS involves requesting, installing and maintaining a certificate. Certificates are requested from a *certificate authority*.

2 URLs differing only by HTTPS instead of HTTP, don't necessarily lead to the same resource.

2.3.2 HTTP/2

HTTP/2 is an update of the HTTP protocol that addresses some key bottlenecks.

TCP connections are costly due to 3-way handshake, the amount of connections per host is now limited. HTTP is very latency-sensitive. *HTTP pipelining* allows issuing multiple requests on a single TCP connection without waiting for responses first.

Many websites used workarounds to circumvent these limitations. Such as embedding scripts and styles into HTML documents, combining several scripts, styles or images into single files or *sharding* which is distributing resources across different domains to bypass connection restrictions.

Therefore, HTTP/2 became a binary protocol that sends frames over multiplexed streams.

In practice, HTTP/2 always goes over TLS, even though the standard allows TCP.

2.3.3 Security

Protocol-level security guarantees privacy only from endpoint-to-endpoint. The Web evolved from a document system to a distributed application platform. There need to be application-level security measures.

2.3.3.1 Code injection

Code injection concerns executing malicious scripts on the *server*.

The cause is improper input validation by the server. It may be defended against by validating input ranges and escaping values before passing to scripts.

A common example used to be SQL injection.

Client-side validation is good to offer usability, a server must always (re-)validate. Client-side validation may be bypassed by changing the HTML at runtime, or using another way to send a HTTP request.

2.3.3.2 Cross-site scripting (XSS)

Cross-site scripting concerns executing malicious scripts on the *client*.

It is caused by improper input validation. It may be defended against by proper validation.

2.3.3.3 Cross-origin access

Cross-origin access concerns accessing information from other websites.

It is caused by the existence of `XMLHttpRequest` objects that're logged into other websites. Browsers block cross-origin requests by default.

This problem is unique to browsers, due to cookies. Browsers add an `Origin` header to requests sent by client-side JavaScript.

Chapter 3

Web APIs

The handle *affords* opening the door. The handle is an *affordance* for opening the door. Affordances strongly affect usage in software. For example, many people can use Word, few Vim. Affordances are of crucial importance on the Web.

A Web API is a server-side HTTP interface that exposes data and functionality to clients. When well-designed, a Web API provides affordances for people and machines to interact with it.

3.1 The REST architectural style

REST stands for **Re**presentational **S**tate **T**ransfer.

3.1.1 The REST constraints

The REST architectural style consists of 5 mandatory constraints;

1. the client-server constraint,
2. the statelessness constraint,
3. the cache constraint,
4. the layered system constraint,
5. the uniform interface constraint.

3.1.1.1 Client-server

REST architectures inherit the client-server constraint from the Web. Client application and server application must be able to evolve seperately.

3.1.1.2 Statelessness

The server does not remember the client, nor any previous requests. Statelessness refers to application state. The server still maintains a resource state.

Visibility Requests contain all context necessary to understand them. Looking at a single request is sufficient to visualize the interaction.

Reliability Since every request is stand-alone, failure of one request doesn't influence success of other requests.

Scalability The server doesn't need to remember application state, enabling it to process more requests in shorter time.

Hypermedia is the **representation** of a resource along with **controls** (links) that lead to next steps.

Statelessness trades off server-side control for bandwidth. Clients are responsible for transitioning from one state to another. They mustn't be trusted to make allowed transitions.

3.1.1.3 Cacheability

A server response may indicate whether it is cacheable or not.

Caching improves network efficiency, scalability and user-perceived performance. Reliability may suffer if cached data becomes stale.

3.1.1.4 Layered system

Intermediaries may be inserted transparently. That means a client cannot tell whether it is connected directly to the end server, or to an intermediary along the way.

Layering increases flexibility but may introduce increased overhead and latency.

3.1.1.5 Uniform interface

Identification of resources Any identifier points to at most 1 resource. A resource may be pointed to by multiple identifiers. If an identifier points to multiple resources, it creates a collision.

Resource manipulation through representation Resources may change over time. It associates a definition with a value. This allows late binding.

Information resources may have zero or more representations. Depending on the needs and capabilities of a client, resources may be represented differently. For example, in Dutch or English, or in HTML or JSON.

Representations themselves may have identifiers. Ruben says that representations become resources when they are identified.

Self-descriptive messages ntermediaries may interpret messages (think: caches). Also, refer back to statelessness.

Response and request bodies are expressed in explicit, agreed-upon media types.

HATEOAS Responses contain hypermedia controls such as hyperlinks and forms. They *afford* the next steps a client may take.

This is the *Web* aspect of REST.

Hypermedia defines REST interactions, minimizing the client-server contract. Client and server may evolve independently. From one resource, the client discovers the interface. The client doesn't bind to an identifier, but is expected to know the semantics of controls.

It is expected from a server to afford clients the controls they need.

It's a good idea to synonymize a resource with a webpage.

3.1.2 REST applied to the Web

Note that the REST style may be applied to non-Web interfaces. It's just that most REST APIs are HTTP APIs.

Constrained Application Protocol (CoAP) is an example of a non-Web REST implementation.

3.1.2.1 Client-server

Clients make HTTP requests to servers. Servers reply with HTTP responses. Note that the client-server is relative to the interaction.

3.1.2.2 Statelessness

HTTP is designed as a stateless protocol. This causes a lot of repetition in header information `Host`, `Accept`, `User-Agent`, ...

3.1.2.3 Cacheability

HTTP responses may be cached, special headers are available for this. HTTP method semantics influence caching, responses to safe methods may be cached.

3.1.2.4 Layered system

HTTP allows transparent insertion of intermediaries. Because of statelessness, it is easy for intermediaries to answer.

3.1.2.5 Uniform Interface

The Web supports all 4 constraints by implementing a combination of URL, HTTP and HTML.

Identification of resources An HTTP URL points to at most one resource. The concept pointed to by a URL shouldn't change.

Manipulation of resources through representations Clients obtain different representations of a resource through *content negotiation*.

A client may indicate their preference with the `Accept` header, a server then advertises its choice with the `Content-Type` header.

Self-descriptive messages The HTTP methods and headers allow for self-description.

There are a limited number of methods. Intermediaries are aware of the semantics of each method.

The set of HTTP headers is extensible. Intermediaries may forward headers they don't understand.

HATEOAS HTML and other hypermedia documents may contain hypermedia controls.

Users shouldn't have to touch the address bar. If they do, the action isn't afforded by hypermedia.

3.2 Sustainability of information

3.2.1 Web APIs with and without REST

A ton of do's and don'ts.

The term *hypermedia API* is sometimes used to indicate APIs that follow all REST principles.

A client should be able to *follow its nose*, from any starting point.

3.2.2 Designing Web APIs for the long term

REST architectures are aimed at sustainability. URLs should remain constant while technologies evolve.

An elaborate example on Europeana is given.

3.3 Web API technologies

3.3.1 Non-REST Web services

Web services typically refer to XML-based interfaces on top of HTTP.

While *Web APIs* is a more fashionable term for interfaces that use HTTP and JSON.

3.3.1.1 Simple Object Acces Protocol (SOAP)

The SOAP transfers XML messages on top of HTTP. Such a message contains an `Envelope` element, which in turn contains `Header` and `Body` elements.

SOAP ignores its underlying protocol, instead it aims at supporting many possible underlying protocols. Thus, it also ignores the benefits of the Web.

All requests use POST, and thus may not be cached. Intermediaries cannot understand requests because SOAP ignores HTTP. Client and server cannot evolve separately, their interaction is "hard-coded".

3.3.2 Web API descriptions

Web API descriptions are *machine-interpretable* details about an API. There exist 3 broad uses for descriptions.

In-band Resource and hypermedia control details.

Interface Structural properties.

Functional Effect and/or purpose.

3.3.2.1 Web Services Description Language (WSDL)

WSDL describes SOAP service structure.

A WSDL description is an XML document that simply lists a service's methods and parameters.

A WSDL description and SOAP wrapper may be generated from program code.

Code to interact with the SOAP service may be generated from the WSDL description.

3.3.2.2 Semantic Markup for Web Services (OWL-S)

OWL-S captures the semantics of single SOAP operations using RDF.

An OWL-S description consists of three parts.

Profile High-level description for advertising and discovery.

Grounding How an client accesses the service.

Model How a client should use the service.

3.3.2.3 Web Application Description Language (WADL)

WADL is the WSDL of non-SOAP Web APIs.

A WADL description is an XML document that lists HTTP methods and parameters as well as representation structure.

WADL also supports automatic code generation.

3.3.2.4 Most recent Web API description formats

OpenAPI documents a Web APIs methods in JSON.

API Blueprint focuses on data modeling.

3.3.2.5 Hypermedia-based formats

Hypermedia-based formats place their description *in* the message

Hydra is an RDF vocabulary to describe hypermedia controls and API structure.

Chapter 4

Metadata and Semantics: Introduction to the Semantic Web

4.1 Introduction

Metadata is structured information that makes it easier to interact with an information resource.

The solution to information overload is more information as long as that information is metadata.

Metadata *used to be* just data about data. Which is already useful for categorization. For example record cards in a library.

Metadata *today* is standardized, structured information which may steer automated processes. Its goal is to make data understandable, i.e. more than just locate it.

Metadata *ultimately* will provide semantic interoperability. It will provide a medium where man and machine, given the same data, may come to the same conclusions.

Metadata is necessary for *persistence* of information. As in the example of a person born in Nieuwdorp, where the name of the village changed to [...], metadata helps identify where the person was actually born.

Metadata may exist *before* the information it will describe. Such as the scripts or scenarios of a movie.

Metadata may exist *after* the information it described. Such as the comments to a deleted video.

Taxonomies were and are used to classify information. Now we prefer *Folksonomies*, annotations in free text, no predefined structure, no hierarchy, users may add metadata collaboratively. An example is a tagcloud.

Tags know some *issues*, such as cultural differences, e.g. for some Genghis

Khan is a hero, for others, a war criminal. Different tags may mean the same thing (due to linguistic differences). A tag may mean different things, e.g. “Apple”. Tags may contain spelling errors.

There exist (many) *metadata standards*. Interoperability requires mappings. Many of the issues of tags carry over. For example, descriptions of a creator may be ”Shakespeare, William”, ”William Shakespeare”, ”Shakespeare”, ”W. Shakespeare”. It is difficult to automatically detect *semantical* errors in metadata. However, it's easy to detect *syntactical* errors.

The current Web is built up almost exclusively syntactically.

4.2 Adding semantics to data (beyond XML)

The *Resource Description Framework (RDF)* is a W3C standard. It represents data in the form of a graph. At the atomic level, it consists of triples.

$$\text{Subject} \xrightarrow{Predicate} \text{Object},$$

where subject, predicate and object are all *URLs*! If they aren't literals. But it's still prettier to think them all URLs.

An *ontology* is formal definition of a set of concepts within a domain and the relationships between those concepts. It is used to *reason* about the properties of that domain and may be used to define the domain.

Concepts may include: classes, instances, attributes, relations, functions, constraints.

4.3 Introduction to the Semantic Web

Machines need access to unambiguous information. This information must be combinable, exchangable and machines must be able to *reason* about it.

There is a necessity for a web of *data*, the Semantic Web, instead of merely a web of *documents*, the Syntactic Web.

It must be integrated into the current Web. It mustn't be a separate (secondary) web.

Data is represented by graphs. In practice, this data is serialized in RDF. RDF can be expressed in several syntaxes. Applications may then be built on top of this data, querying and manipulating it as necessary.

Information is abstracted as such because

- the graph format is independent of data format,
- changes in the schema of a local database don't prompt changes throughout the entire data,
- it is trivial to add new information and connections.

4.4 Open Linked Data: the Web of data

4.5 So what?