2nd Edition

Tale of Two Systems

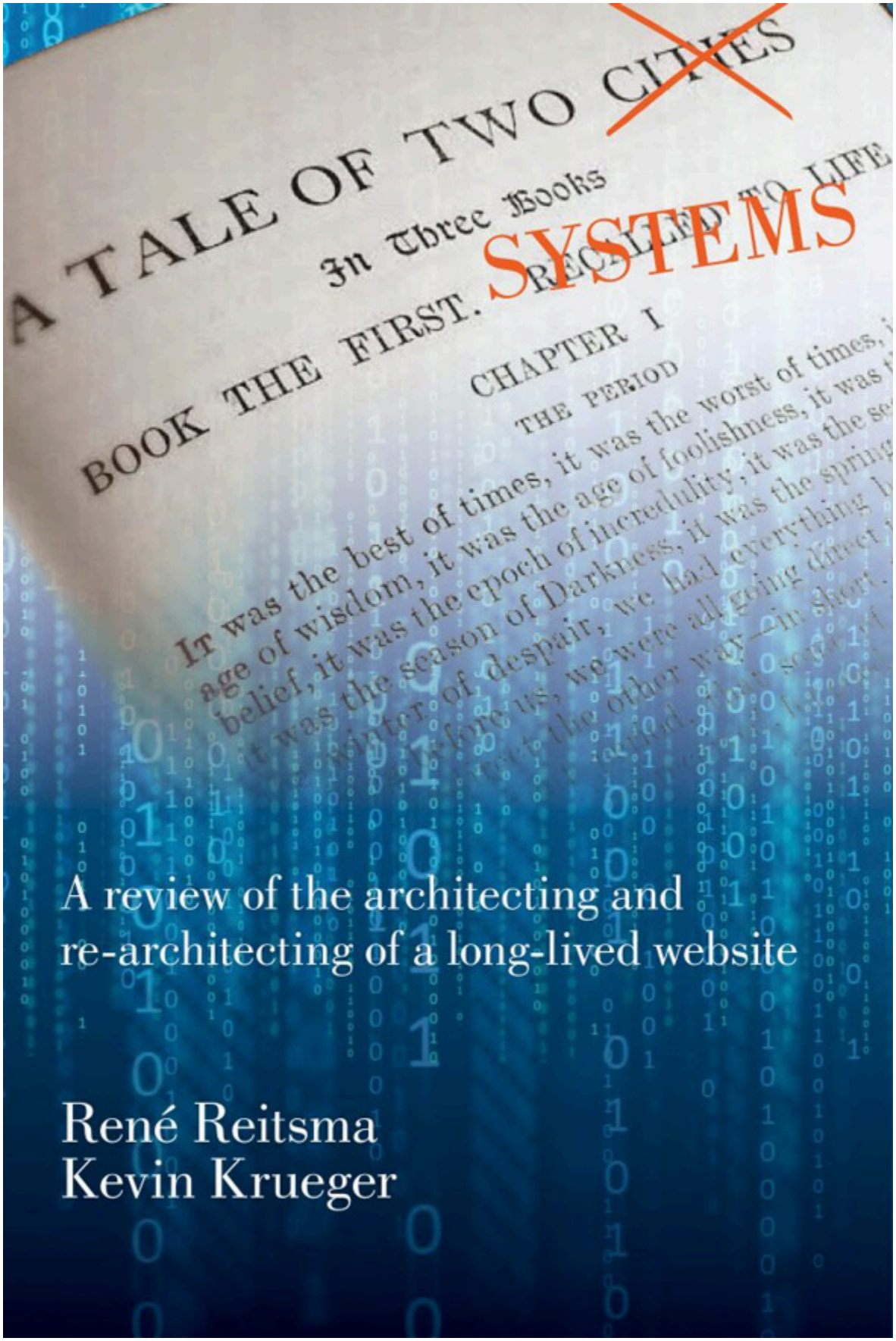
A Review of the Architecting and Re-Architecting of a Long-Lived Website

René Reitsma and Kevin Krueger

Open Educational Resources

Oregon State University

A Tale of Two Systems



A review of the architecting and re-architecting of a long-lived website

René Reitsma
Kevin Krueger

Publication and ongoing maintenance of this textbook is possible due to grant support from Oregon State University Ecampus.

Suggest a correction (bit.ly/33cz3Q1)

Privacy (open.oregonstate.education/privacy)

This book was produced with Pressbooks (https://pressbooks.com) and rendered with Prince.

Contents

Preface

One of the hallmarks of successful information system architectures is their longevity. Luc Hohmann's insightful 2005 book *Beyond Software Architecture* carries this notion in its subtitle: *Creating and Sustaining Winning Solutions*. Coming back to school from their internships in industry, our students often comment on how old some of the information systems they worked with that summer were. In times where the media keep telling us that current technology will be obsolete in six months, these old systems must indeed appear anachronistic; fossils from a time when people wrote their programs in COBOL on green screens.

Older, so called 'legacy' systems or applications, however, survive for any number of reasons. Some of these reasons are not very desirable; for instance, the lack of agility of an organization, users' unwillingness to learn new things, or the conservative power of those who have sold their hearts to the old system. Yet good system architectures are good precisely because they have been designed to accommodate the vagaries of a changing world; *i.e.*, they can be adapted and extended so that they do indeed 'live long.' So long, in fact, that they may survive their original designers.

Still, regardless of how good systems serve their users, eventually, when the cost for fixes, adaptions and extensions becomes too high or when new technologies offer opportunities for providing entirely new services or significant efficiencies and speed gains, it is time to either buy, rent or build something new.

This is the story of one such rebuilds. The system in question is www.teachengineering.org —TE— a digital library of K-12 engineering curriculum that was built from the ground up with established technology and which for 13 years (2002—2016) enjoyed lasting support from its growing user community and its sponsors. Those 13 years, however, covered the period during which smartphones and tablets became commonplace, during which the Internet of Things started replacing the Semantic Web, during which NoSQL databases made their way out of the research labs and into everyday development shops, during which we collectively started moving IT functions and services into 'the cloud,' and during which computing performance doubled a few times, yet again. Alongside these technical developments we saw a rapidly growing emphasis on usability and graphic design, partly because of the need to move applications into the mobile domain, and partly because of the need and desire to improve both ergonomics and aesthetics. During this same period, TeachEngineering's user base grew from a few hundred to more than 3 million users annually; its collection size quadrupled; it went through several user interface renewals, and significant functionality was added while having an exemplary service record, and it enjoyed continued financial support from its sponsors. All of this took place without any significant architecture changes. In Hohmann's terms, it was indeed a 'winning architecture.'

Yet, although the system architecture could probably have survived a while longer, it started to become clear that with the newer technologies, better and newer services could be developed faster and at lower cost, that moving most of its functionality into the cloud would both boost performance and lower maintenance cost, and that the system's resource and code footprint could be significantly reduced by rebuilding it on a different architecture, with different and more modern technology. And, of course, with a new architecture some of the mistakes and unfortunate choices built into the old architecture could be avoided.

In this monograph we provide a side-by-side of this rebuild. We lay out the choices made in the old architecture —we refer to it as *TE 1.0*— and compare and contrast them with the choices made for *TE 2.0*.

We explain why both the 1.0 and 2.0 choices were made and discuss the advantages and disadvantages associated with them. The various technologies we (briefly) describe and explain can all be found in traditional *Information Systems Design & Analysis* textbooks. However, in the traditional texts these technologies are typically presented as a laundry list of options, each with advantages, disadvantages and examples. Rarely, however, are they discussed in the context of a single, integrated case study and even rarer in an evolutionary and side-by-side, 1.0 *vs.* 2.0 fashion.

The two systems, TE 1.0 and TE 2.0, both share as well as use alternate and contrasting technologies. Both groups of technologies are discussed, demonstrated and compared and contrasted with each other and the reasons for using them and/or replacing them are discussed and explained.

Along with the discussion of some of these technologies, we provide a series of small cases from our TeachEngineering experience, stories of the sort of things system maintainers are confronted with while their systems are live and being used. These range from strange user requests to Denial of Service attacks and from having to filter out robot activity to covert attempts by advertisers to infiltrate the system. For each of the technologies and for most of these case histories we provide —mostly on-line— exploratory exercises.

Intended Audience

This text is meant as a case study and companion text to many *Information Systems Analysis & Design* textbooks used in undergraduate Management Information Systems (MIS), Business Information Systems (BIS) and Computer Information Systems (CIS) programs. The US counts about 1,300 (undergraduate + graduate) such programs (Mandiwalla *et al.*, 2016). These texts typically contain short descriptions of technologies which give students some sense of what these technologies are used for, but do not provide much context or reflection on why these technologies might or might not be applied and what such applications actually amount to in real life. As a consequence, students, having worked their way through these textbooks and associated courses will have had little exposure to the reasoning which must take place when making choices between these technologies and to what goes into combining them into working and successful system architectures. It is our hope that this *Tale of Two Systems* (pun very much intended) will help mitigate this problem a little.

Instructor Support

Although instructors of *Systems Analysis and Design* courses typically have themselves experience in architecting and building systems, setting up demonstration and experiential learning sites equipped with cases where students can explore and practice the technologies discussed in textbooks and coursework can be daunting. Fortunately, technology-specific interactive web sites where such exploration can take place are rapidly becoming available. Examples of these are w3schools.com, sqlfiddle.com, dotnetfiddle.net and others. In the exercise sections of our text (all marked with the  symbol) we have done our best to rely

on those publicly available facilities where possible, thereby minimizing set-up time and cost for instructors as well as students.

In cases where we could not (yet) rely on publicly available services, we have included instructions on how to set up local environments for practicing; most often on the reader's own, personal machine.

This book does not (yet) contain lists of test and quiz questions or practice assignments for two reasons. First, we believe that good instructors can and are interested in formulating their own. Second, we have not have had the time to collect and publish those items. However, we very much do invite readers of this text to submit such items for addition to this text. If you do so (just contact one of us), we will take a look at what you have, and if we like it we will add it to our text with full credits to you.

References

- Hohmann, L. (2005) *Beyond Software Architecture*. Creating and Sustaining Winning Solutions. Addison Wesley.
- Mandiwalla, M., Harold, C., Yastremsky, D. (2016)

  Information Systems Job Index 2016. Assoc. of Information Systems and Temple University. http://isjobindex.com/is-programs

Preface to the Second Edition

This second edition of *Tale of Two Systems* comprises an update of the original text. Several factors compelled this update:

- Quite a few of the links in the original text had become invalidated. Some of these could simply be updated, but others were pointing to places which have disappeared since the publication of the first edition of this text. The locations in the text where that applied had to be accordingly modified.
- Some of the places on the web which in the first edition were suggested as good places to practice and run coding exercises have disappeared.
- Some of the first-edition exercises can no longer be run on these public places because of security reasons.
- The first edition contained exercises in C# and PHP and used both RavenDB and MongoDB for NoSQL exercises. Not only does maintenance of instructions for each of these constitute quite a bit of work, but students of the text had to install and work with multiple platforms. Hence, we decided to change all exercises to use only Python and MongoDB.
- One of the appendices was removed and its contents integrated in one of the regular chapters.
- A new exercise was added to Chapter 2.
- A chapter on a recently added tool —The NGSS *Explorer*— and its architecture was added.
- A large number of editorial changes were made.
- References to new tools and texts were added.

About the Authors

René Reitsma

René Reitsma is a professor of Business Information Systems at Oregon State University's College of Business. He grew up and was educated in the Netherlands. Prior to receiving his PhD from Radboud University in 1990, he worked at the International Institute for Applied Systems Analysis (IIASA) in Laxenburg, Austria on a project developing an expert system for regional economic development. René joined the University of Colorado, Boulder's Center for Advanced Decision Support in Water and Environmental Systems (CADSWES) in 1990, working on the design and development of water resources information systems. In the summer of 1998 he accepted a faculty position in Business Information Systems at Saint Francis Xavier (STFX) in Nova Scotia, Canada. He joined the faculty at Oregon State University in 2002.

René teaches courses in Information Systems Analysis, Design and Development and has ample practical experience in designing and building information systems. His motivation for writing this book was the lack of technical exercises, examples and critically discussed experiences of real-world system building in the typical Information Systems Design and Development textbooks.

René can be reached at *reitsmar 'at' oregonstate.edu*

Kevin Krueger

Kevin Krueger is Founder and Principal Consultant at SolutionWave, a boutique custom software development firm. With more than 15 years of software development experience, he currently specializes in web application development. He enjoys the exposure to a wide variety of industries that being a consultant affords him. Depending on the client, he acts in a number of roles including software developer, product manager, project manager, and business analyst. He has worked on projects for Fortune 500 organizations, large public institutions, small startups, and everything in between.

Kevin's business experience started in the 9th grade when he and a business partner started selling computers in rural North Dakota. A few years after graduating from North Dakota State University in Fargo with a B.S. in Computer Science, Kevin moved to Colorado in 2001 where he currently resides with wife and daughter.

Kevin can be reached at *solutionwave.net*

Both René and Kevin are TeachEngineering.org (TE) developers. Whereas René was responsible for the design and development of the first version of TE (TE 1.0), Kevin is the designer and developer for TE 2.0 which runs on a different set of technologies and a different system architecture. Since they both are interested in making good architectural decisions, and since they both believe that much can be learned from comparing and contrasting TE 1.0 with TE 2.0, they decided to collaborate on writing this text.

Acknowledgements

This material is based upon work supported by the National Science Foundation under grant no. EEF 1544495. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1. TeachEngineering (TE) Overview

Introduction

This chapter contains an overview of the TeachEngineering (TE) system (www.teachengineering.org). We briefly discuss its origin as an attempt to build a unified digital library from a diverse set of K-12 engineering curricula which had been developed at various US engineering schools. At the time of its inception in 2002, few if any standard solutions for this problem were available and the system was designed and developed from the ground up. It became one of several hundred digital libraries which around 2005 comprised the *National Science Digital Library* (NSDL) project, funded by the US National Science Foundation (Zia, 2004). Now, quite some time later, TeachEngineering enjoys a user base of about 3 million users, has recently been rebuilt on a new architecture and remains as one of only about 35 collections in NSDL.

Brief History

In the late 1990s the Division of Graduate Education (DGE) of the US National Science Foundation started its *Graduate STEM Fellows in K-12 Education* or GK-12 program (NSF-AAAS, 2013). The program meant to bring K-12 Science, Technology, Engineering and Mathematics (STEM) teachers and university graduates together in an attempt to develop new and innovative K-12 STEM curriculum. With an annual budget of about $55 million, by 2010 the program had made 299 awards at 182 institutions, provided resources to almost 11,000 K-12 teachers in 5,500 schools and impacted more than 500,000 K-12 students (NSF, 2010). In the process, a treasure trove of innovative STEM curriculum had been developed.

Unfortunately, most of that curriculum sat undetected on 'Google shelves' where, through the usual processes of web site entropy and link rot, it quickly became stale and abandoned as its creators, having completed their projects, went their ways. It was, as described by Lima (2011) in relation to information visualization websites, as if "*the whole field suffers from memory loss.*" To make matters worse, the GK-12 curriculum was not published to any standards and hence, existed in a multitude of layouts and electronic formats, various logical structures and hierarchies, and followed many different kinds of pedagogical approaches. As a consequence, much GK-12 curriculum was difficult to find; was lost in a sea of Google search results; was rarely aligned with K-12 educational standards; was not maintained; and was difficult to compare with and relate to other GK-12 materials.

Foreseeing this process of 'curricular entropy,' a group of engineering faculty and GK-12 grant holders at the University of Colorado, Duke University, Colorado School of Mines, and Worcester Polytechnic University, supplemented by one of us from Oregon State University (Figure 1) decided to apply for an NSDL grant to try and collect all GK-12 engineering curriculum —the E in STEM— standardize it, make it searchable, align it with K-12 educational standards, and host it as one of NSDL's digital libraries. That was in 2002. Now, 20 years later, TeachEngineering is still supported by NSF and a variety of other funders. It has grown to more than 1,800 curricular resources contributed by many universities and programs

and enjoys a patronage of about 3 million users per year. NSF's GK-12 program is no longer active but curriculum developed in other NSF programs, such as Research Experiences for Teachers (RET) and Math and Science Partnership (MSP), as well as that by other K-12 curriculum developers continues to find its way to TeachEngineering for two reasons: because NSF encourages its grantees to publish their work in TeachEngineering and because TeachEngineering consolidates good K-12 Engineering curriculum in a single, searchable and easily operable location on the web.

Figure 1: *TeachEngineering's original developers from the University of Colorado, Duke University, Colorado School of Mines, Worcester Polytechnic University and Oregon State University.*

The TeachEngineering Resource Collection

A quick look at the TeachEngineering curriculum (https://www.teachengineering.org/curriculum/browse) reveals how the library is structured. It consists of a collection of resources, each of which is of one of five types: *activity*, *lesson*, *curricular unit* , *maker challenges* and *informal learning activities*. *Lessons* introduce certain topics and provide background information on those topics. Practically all lessons refer to one or more *activities* which are hands-on exercises that illustrate and practice the concepts introduced in the lesson. When several lessons all address a central theme, they are often —although not necessarily— bundled in a so-called *curricular unit*. *Informal learning activities* are shortened versions of activities. They

are not part of lessons and curricular units and are therefore not part of fully structured learning plans. Instead, they are meant for informal learning settings such as after-school clubs. All informal learning activities are in both English and Spanish. Finally, *maker challenges* are open ended design challenge prompts. Whereas *activities* are quite prescriptive, *maker challenges* are meant to maximize design freedom and project timelines.

All resources are further classified to belong to one or several of 15 subject areas (Algebra, Chemistry, Computer Science, *etc.*). For example, the unit *Evolutionary Engineering: Simple Machines from Pyramids to Skyscrapers* is categorized under the subject areas *Geometry*, *Physical Science*, *Problem Solving*, *Reasoning and Proof*, and *Science and Technology*. It has six lessons and seven activities. Lessons and activities do not have to be part of units; they can live 'on their own,' and activities can be part of curricular units without being part of a lesson. This hierarchical structure is strictly unidirectional. This means that whereas activities or lessons can live on units, units cannot live on a lesson or an activity. Nor can a lesson live on an activity (Figure 2).

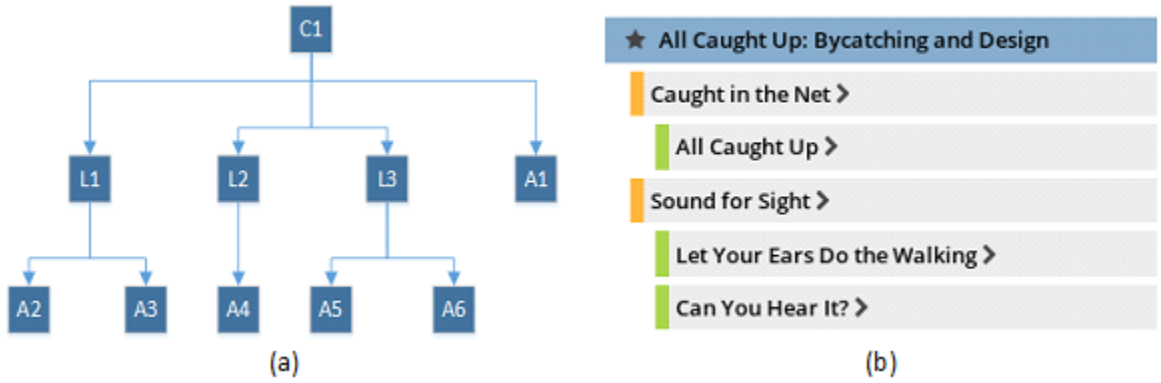


Figure 2: *Hierarchical structure of TE documents. (a) General example. The curricular unit C1 has three lessons (L1-L3) and six activities (A1-A6). Five activities (A2-A6) reside on lessons and one (A1) resides directly on the unit. (b) The All Caught Up curricular unit consists of two lessons and three activities.*

As of June 28, 2022, TeachEngineering had 110 curricular units, 524 lessons and 1,125 activities. In addition to these hierarchically structured resources, TeachEngineering also contains sets of stand-alone learning resources: 41 *maker challenges* and 33 *informal learning activities*.

Controlled Resource Content

In order for TeachEngineering to expose and disseminate K-12 engineering curriculum in a single, unified, standards-aligned, searchable and quality-controlled digital library, a standardized structure was imposed on each and every resource. For instance, all activities, lessons and curricular units must have a summary section, a section which explains the curriculum's connection with engineering, a set of keywords, the document's intended grade level, the time required to execute it, a set of K-12 educational standards to which the curriculum is aligned, *etc.*

Figure 3 shows part of a TeachEngineering activity as it appears in a user's Web browser. Note its *Summary*, *Engineering Connection* and the data in the *Quick Look* box.

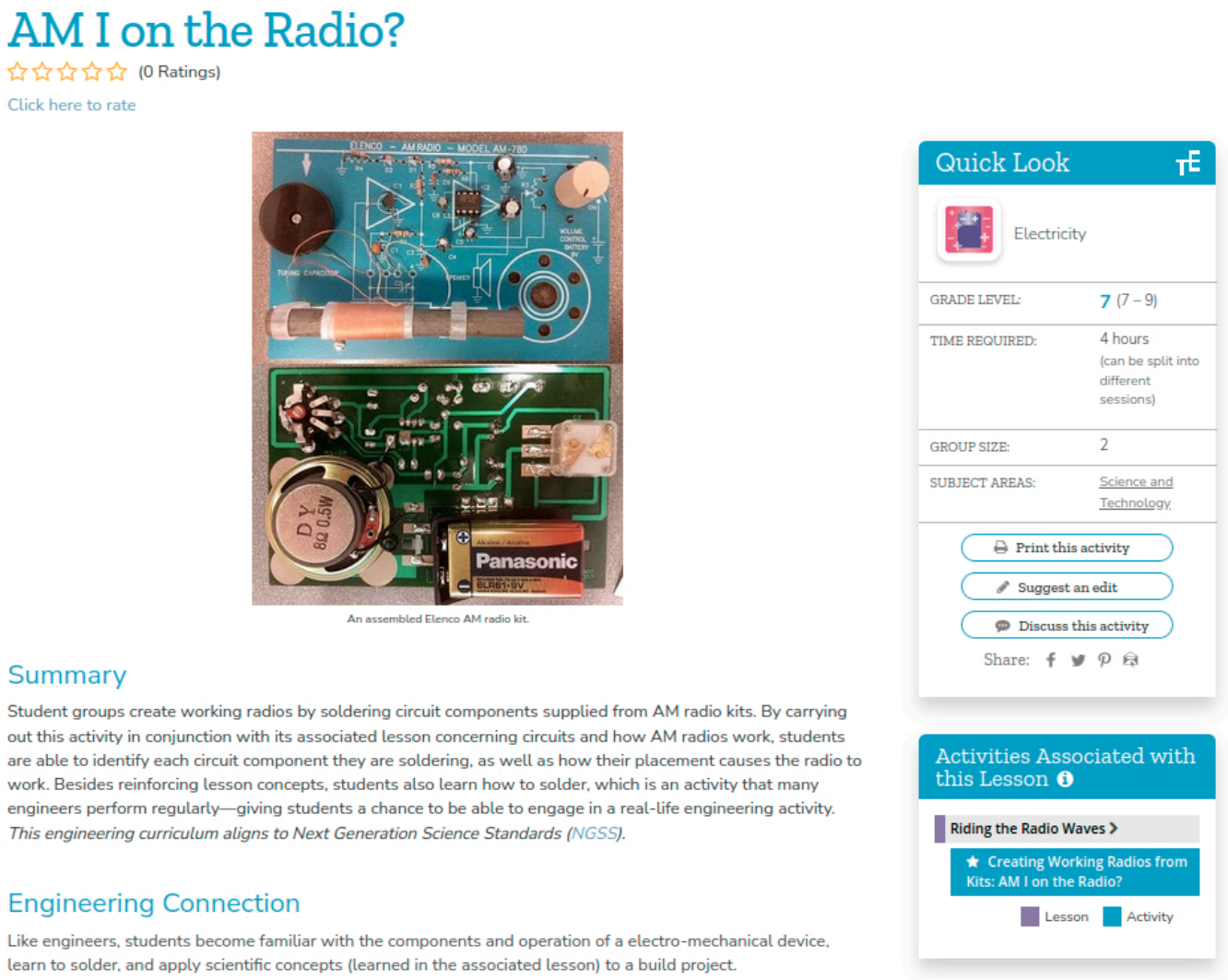


***Figure 3**: partial rendering of a TeachEngineering activity.*

Although the precise list of document components —different for different resource types— is not important here, it is important to realize that these components come in two types: mandatory and optional. Figure 4 graphically displays some of the components of a TeachEngineering lesson. Solid-line rectangles indicate mandatory components, while dashed lines indicate optional components. Note that the content specification once again is a hierarchy. For instance, while a lesson must have a grade specification (*te:grade*), the grade specification itself must have a *target* with optional *upper*– and/or *lower* bounds. When we consider this type of hierarchy as a tree (Figure 4), the leaves of this tree —its terminal nodes— must be of a specific computational data type (not shown in Figure 4). For instance, the *target lowerbound* and *upperbound* values must be integers.

Putting such strict structural and data type constraints on resource content accomplishes two things. First, it allows the construction of a collection of resources with a single, unified content structure and a

common look-and-feel. Second, and just as important, it allows for automatic, software-based procedures for processing collection content. Examples of these processes are resource ingestion and registration —a process known as *resource accessioning*— quality control, resource indexing, and metadata generation.

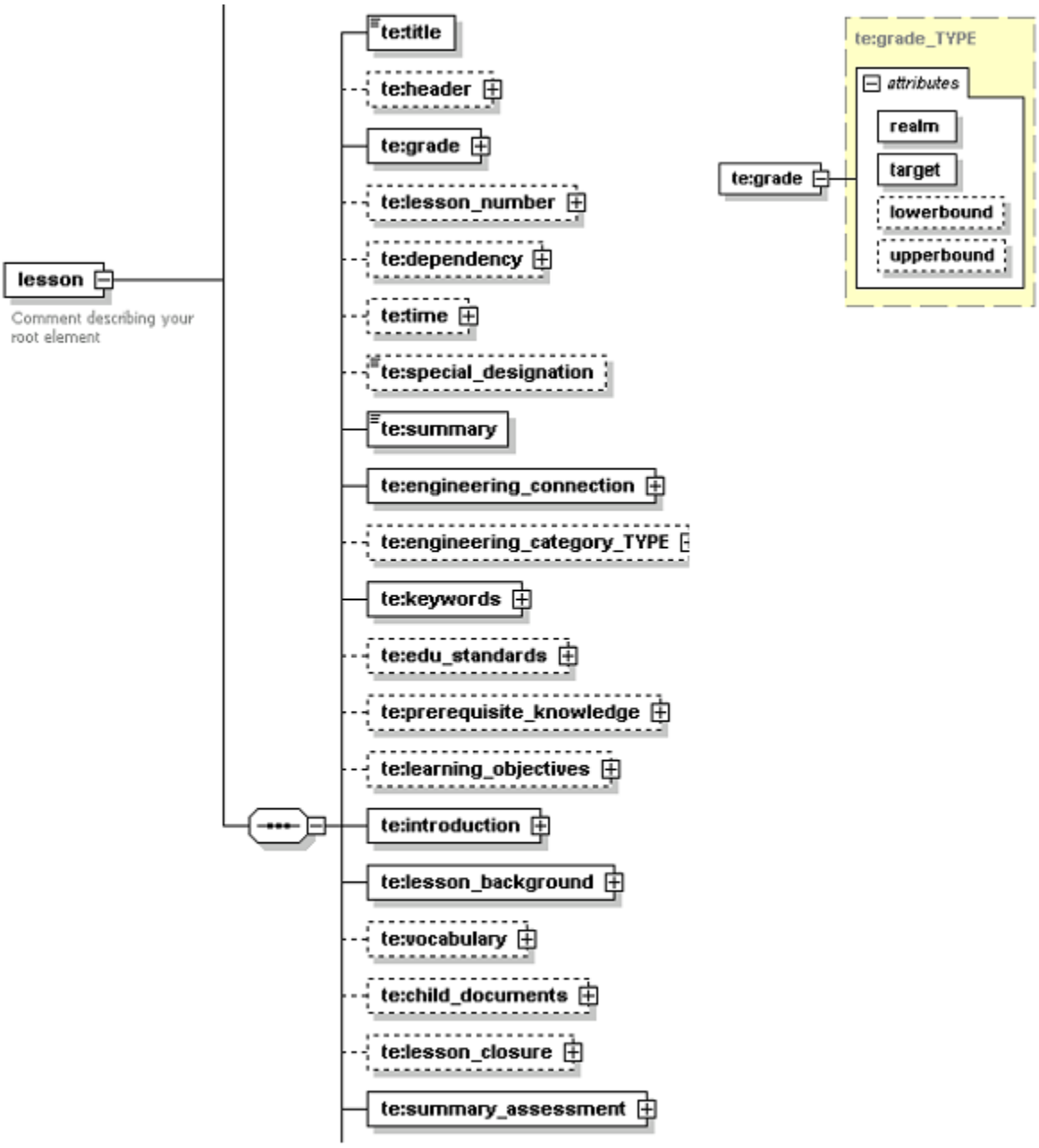


***Figure 4**: Partial structure and content of a lesson. Solid-line rectangles indicate mandatory components; dashed lines indicate optional components (graphic generated with XMLSpy).[footnote]The resource structure as displayed here is that of TE 1.0. In TE 2.0 some of these components were dropped and others were added. The principle that resources have an enforced structure, however, remains unaltered.[/footnote]*

The latter point —automatic generation of metadata— is important as it is a classic bottleneck in the

publication and dissemination of digital resource repositories. With metadata, we mean data about a resources rather than the resources' content itself. For example, author names, copyrights and title, or in our case, things such as grade level, keywords, expendable cost, required time, *etc*. Collections which rely on manually entered metadata often suffer from the problem that manually entering such data is boring, time consuming, is error prone and must be reviewed and possibly redone when a resource changes or disappears. As a consequence, much of the items in digital libraries have very minimal metadata, which itself reduces their chances of being found in searches and therefore their chances of being used. Being able to automatically generate good metadata is therefore a good thing. Chapters 3, 4 and 5 cover the various ways in which TE content was and is controlled in TE 1.0 and 2.0.

2022 update. At this time —June 2022— the 'lack of metadata' problem remains. As an example we mention the collection of NextGenScience Quality Examples, a set of 400+ digitally available STEM curriculum resources which have been lauded for their use of and alignment with the Next Generation Science Standards. Whereas the resources in this collection are of very good quality, they are difficult to find as a collection and they are impossible to search. Why? Because nobody has bothered to index them and to publish their metadata. Each and every one of the 400+ resources has structured content and is located somewhere on the Internet, yet nowhere is there a publicly, machine-accessible index of all these resources. We revisit this topic in Chapter 3.

K-12 Educational Standards

It will come as no surprise that in this age of performance metrics, K-12 education is subject to delivering predefined learning outcomes. These outcomes are known as 'educational standards' or 'performance expectations.' In the USA the determination and authoring of these standards is the authority of individual states and within the states sometimes that of individual districts. Although attempts at harmonizing standards across states have met with some success —we refer to the Common Core for Mathematics and English and the Next Generation Science Standards initiatives (Conley, 2014; NRC, 2011)—, the USA educational standards landscape remains quite complex and in constant flux. Not only do standards change on a regular basis, but different states have different standards; they completely or partially adopt the harmonization standards, they write their own variations of those standards or they write standards of their own. When they write their own, the standards show great variety in granularity and approach between states (Marshall & Reitsma, 2011; Reitsma & Diekema, 2011; Reitsma *et al.*, 2012). The fact that across the whole USA there are over many thousands of K-12 science standards, speaks to the complexity of the USA's K-12 standards landscape.

Early in development of TE 1.0, our TeachEngineering team decided that we were in no position to continuously track evolving K-12 Science and Engineering standards and that we would prefer acquiring those standards from an external provider. Similarly, since figuring out for all 50 states and for each resource which standards are supported by (align with) which resource was not one of our core competencies, we hoped to acquire this service from another party.

Fortunately, at the time of TE 1.0 development, the *Achievement Standard Network (ASN)* project was established (Sutton & Golder, 2008) and was, like TeachEngineering, partly funded by NSF's NSDL project. The ASN project maintains a repository of all K-12 educational standards in the USA, issues new versions

when standard sets change and makes these sets available to others. The ASN is currently owned and operated by D2L, a learning-platform company. TeachEngineering has enjoyed a long-term relationship with ASN and although major differences exist between how TE 1.0 used to and TE 2.0 now uses the ASN, the ASN continues to function as TeachEngineering's *de facto* repository of K-12 standards.

Whereas K-12 standard tracking is readily available from services such as ASN, standard alignment; *i.e.*, matching learning resources with standards, is far more problematic. Although it is not our goal here to thoroughly analyze the standard alignment problem, it is important to know that in TeachEngineering each resource is aligned with one of more K-12 educational standards and since standards frequently change, these alignments must be regularly updated as well. Chapter 5 covers the differences between TE 1.0 and 2.0 in how they represent and include standards.

Collection Editing and Resource Accessioning

Although invisible to TeachEngineering users, TeachEngineering resources do not make it automatically and miraculously into the collection. Instead, they go through a structured review process, first for content and once accepted for content, for editing and formatting. Once a resource is ready, it must be ingested into and registered to the collection.

The process starts with TeachEngineering curricular staff and several external reviewers reviewing the submissions for content and compliance on standard TeachEngineering acceptance criteria. Is it engineering? Is it good science? Does it fit the objectives of TeachEngineering? Is it well written? Is it aligned with standards and do the alignments seem reasonable? Is it attractive, both for teachers and students? Is it internally consistent? Are concepts properly explained and procedures well described? Is anything missing? Depending on the answers to these questions, a submission might be refused, conditionally accepted, or accepted as is.

Once accepted, the resource must be formatted to fit the TeachEngineering resource template. This process changed quite dramatically between TE 1.0 and 2.0. Finally, the now formatted resource must be registered to the collection so that it can be searched, displayed, saved as a bookmark, commented on, rated, *etc*. Chapter 6 covers resource accessioning in both the TE 1,.0 and TE 2.0 versions.

System Implementation and Collection Hosting

Just as invisible to TeachEngineering users as the collection editing and document accessioning process, is the collection's web hosting. Since TeachEngineering is a web-based digital library it must be web hosted and although the general principles of web hosting are the same for both versions 1.0 and 2.0, the specific implementations are quite different. Whereas TE 1.0 was hosted on Linux on an intra-university Linux cloud, TE 2.0 is hosted on Microsoft's Azure cloud. And whereas TE 1.0 used Google's Site Search Engine, TE 2.0 uses Microsoft's Azure Search and whereas TE 1.0's website was written in the PHP programming language running against a MySQL relational database, 2.0 was written in C# running against the *RavenDB* JSON

database. These differences are significant and reflect some of the more general developments in web-based technologies as they occurred over the last few years. We discuss and illustrate these elsewhere in our text.

Extras

In addition to the TE core functions mentioned above, a system such as TE has a number of what we would like to call 'extras;' facilities which make life for both the system maintainers and the system users easier. Some of these are the following:

- Facilities where users can store frequently used curriculum or where they can rate and review curriculum.
- Facilities for users to contact the collection maintenance staff; for instance, to report a problem or to ask a question.
- Facilities for the staff to track system use.
- Facilities which aggregate collection information so that at any time TeachEngineering staff can know how many documents of type *x* or from school *y* are stored in the collection.
- Facilities which make it easy for users to group and print curriculum.

Once again, both TE 1.0 and 2.0 had/have these (and other) facilities, but they architected them in different ways.

Continuous Quality Control

A system such as TeachEngineering is on-line and is meant to serve users anywhere in the world at any time. But it is also continuously changing in that new resources come on-line, existing ones are revised, new users register themselves, and users who care and desire to do so can leave comments and ratings behind. Add to that that TE has a few million (different) users per year and it becomes clear that a system such as this must be carefully monitored to make sure that it functions properly. And in case it stops functioning properly, technical staff must be alerted and informed about what is wrong so that they can fix the problem. Although both TE 1.0 and 2.0 employed significant amounts of monitoring, they go about it in different ways.

References

Conley, D.T. (2014) *The Common Core State Standards: Insight into Their Development and Purpose*.

Lima, M. (2011) *Visual Complexity. Mapping Patterns of Information*. Princeton Architectural Press. New York, New York.

Marshall, B. Reitsma, R. (2011) World *vs*. Method: Educational Standard Formulation Impacts Document Retrieval. *Proceedings of the Joint Conference on Digital Libraries* (JCDL'11), Ottawa, Canada.

NRC (2012) A *Framework for K-12 Science Education. Practices, Crosscutting Concepts, and Core Ideas*. National Academies Press, Washington, D.C.

NSF (2010) *GK-12: Preparing Tomorrow's Scientists and Engineers for the Challenges of the 21st Century*. Available: http://www.gk12.org/files/2010/04/2010_GK12_Overview.ppt.

NSF-AAAS (2013) *The Power of Partnerships. A Guide from the NSF Graduate Stem Fellows in K–12 Education (GK–12) Program*. Available: http://www.gk12.org/files/2013/07/GK-12_updated.pdf

Reitsma, R., Diekema, A. (2011) Comparison of Human and Machine-based Educational Standard Assignment Networks. *International Journal on Digital Libraries*. 11. 209-223.

Reitsma, R., Marshall, B., Chart, T. (2012) Can Intermediary-based Science Standards Crosswalking Work? Some Evidence from Mining the Standard Alignment Tool (SAT). *Journal of the American Society for Information Science and Technology*. 63. 1843-1858.

Sutton, S.A., Golder, D. (2008) Achievement Standards Network (ASN): An Application Profile for Mapping K-12 Educational Resources to Achievement Standards. *Proceedings of the International Conference on Dublin Core and Metadata Applications*, Berlin, Germany.

Zia, L. (2004) The NSF National Science, Technology, Engineering and Mathematics Education Digital Library (NSDL) Program. *D-Lib Magazine*, 2, 10:3.

2. Why Build (Twice!) Instead of Buy or Rent?

Build, Buy or Rent?

Not very long ago, in-house or custom building of IS application systems, either from the ground up or at least parts of them, was the norm. Nowadays, however, we can often acquire systems or many of their components from somewhere else. This contrast —build *vs.* buy— is emphasized in many modern IS design and development texts (*e.g.*, Kock, 2007, Valacich *et al.*, 2016). Of course, the principle of using components built by others to construct a system has always been followed except perhaps in the very early days of computing. Those of us who designed and developed applications 20 or 30 years ago already did not write our own operating systems, compilers, relational databases, window managers or indeed many programming language primitives such as those needed to open a file, compare strings, take the square root of a number or print a string to an output device.[1] Of course, with the advances in computing and programming, an ever faster growing supply of both complete application systems and system components has become available for developers to use and integrate rather than to program themselves into their systems.

What does this mean in practice? On the system level it means that before we decide to build our own, we inventory the supply of existing systems to see if a whole or part-worth solution is already available and if so, if we should acquire/use it rather than build our own. Similarly, on the subsystem level, we look for components we can integrate into our system as black boxes; *i.e.*, system components the internal workings of which we neither know nor need to know. As long as these components have a usable and working *Application Program Interface* (API); *i.e.*, a mechanism through which other parts of our system can communicate with them, we can integrate them into our system. Note that whereas even a few years ago we would deploy these third party components on our own local storage networks, nowadays these components can be hosted elsewhere on the Internet and even be owned and managed by other parties.

Therefore, much more than in the past, we should look around for existing products and services before we decide to build our own. There are at least two good reasons for this. First, existing products have often been tested and vetted by the community. If an existing product does not perform well —it is buggy, runs slowly, occupies too much memory, *etc.*— it will likely not survive long in a community which scrutinizes everything and in which different offerings of the same functionality compete for our demand. The usual notion of leading *vs.* bleeding edge applies here. Step in early and you might be leading with a new-fangled tool, but the risk of having invested in an inferior product is real. Step in later and that risk is reduced —the product has had time to debug and refine— but gaining a strategic or temporary advantage with that product will be harder because of the later adoption. Since, when looking for building blocks we tend to

1. One of us actually wrote a very simple window manager in the late 1980s for MS-DOS, Microsoft's operating system for IBM PCs and like systems. Unlike the more advanced systems at the time such as Sun's SunView, Apple's Mac System, and Atari's TOS, there was no production version of a window managing system for MS-DOS. Since our application at the time could really benefit from such a window manager, we wrote our own (very minimal) version loosely based on Sun's Pixrect library.

care more about the reliability and performance of these components than their novelty, a late adoption approach of trying to use proven rather than novel tools, might be advisable. A good example of purposeful late adoption comes from a paper by Sullivan and Beach (2004) who studied the relationship between the required reliability of tools and specific types of operations. They found that in high-risk, high-reliability types of situations such as the military or firefighting, tools which have not yet been proven reliable (which is not to say that they are unreliable!), are mostly taboo because the price for unreliability —grave injury and death— is simply too high. Of course, as extensively explored by Homann (2005) in his book '*Beyond Software Architecture*,' the more common situation is that of tension between software developers —Homann calls them 'tarchitects'— which care deeply about the quality, reliability and aesthetic quality of their software, and the business managers —Homan calls them 'marketects'— who are responsible for shipping and selling product. Paul Ford (2015), in a special *The Code Issue* of Bloomberg Business Week Magazine also explores this tension.

The second reason for using ready-made components or even entire systems is that building components and systems from scratch is expensive, both in terms of time and required skill levels and money. It is certainly true that writing software these days takes significantly less time than even a little while ago. For example, most programming languages these days have very powerful primitives and code libraries which we can simply use in our applications rather than having to program them ourselves. However, not only does it take (expensive) experience and skills to find and deploy the right components, but combining the various components into an integrated system and testing the many execution paths through such a system takes time and effort. Moreover, this testing itself must be monitored and quality assured which increases demands on time and financial resources.

Nowadays, a whole new dimension of the 'build or buy' issue has been added: security. Sadly, an ever growing army of ill-willing, malevolent actors is continuously attacking our systems, trying to steal information, use our machines as bridgeheads for attacks on other machines, submit us to blackmail, spy on us or hurt us just for the fun of it. This means that we must spend increasing amounts of resources on protecting our digital assets from these people and their evil. In fact, when we think of this, we quickly realize that the massive investments in cybersecurity in the last 10 years or so are almost entirely zero 'value-add,' meaning that these are costs which are incurred but which do not add value to a product. This makes one wonder how much more we could have accomplished if we did not have had to 'waste' all these resources on protecting us from the 'bad guys.'

Regardless, cybersecurity is an everyday issue in today's computing and must therefore be accounted for in de development of our systems. From the 'build or buy' perspective this means that we either must ourselves build safety into our programs (build), or we must trust that such safety is built into the services and products we acquire from others (buy).

So Why Was TeachEngineering Built Rather Than Bought… Twice?

When TE 1.0 was conceived (2002), we looked around for a system which we could use to store and host the collection of resources we had in mind, or on top of which we could build a new system. Since TE was supposed to be a digital library (DL) and was funded by the DL community of the National Science

Foundation, we naturally looked at the available DL systems. At the time, three more or less established systems were in use (we might have missed one; hard to say):

- DSpace, a turnkey DL system jointly developed by Massachusetts Institute of Technology and Hewlett Packard Labs,
- Fedora (now *Fedora Commons*), initially developed at Cornell University, and
- Perseus, a DL developed at Tufts University.

Perseus had been around for a while; DSpace and Fedora were both new in that DSpace had just been released (2002) and Fedora was fresher still.

Assessing the functionality of these three systems, it quickly became clear that whereas they all offered what are nowadays considered standard DL core functions such as cataloging, search and metadata provisioning, they offered little else. In particular, unlike more modern so-called content and document management systems, these early DL systems lacked user interface configurability which meant that users (and those offering the library) had to 'live' within the very limited interface capabilities of these systems. Since these were also the days of a rapidly expanding world-wide web and a rapidly growing toolset for presenting materials in web browsers, we deemed the rigidity and lack of flexibility and API's of these early systems insufficient for our goals.[2]

Of course, we (TeachEngineering) were not the only ones coming to that conclusion. The lack of user interface configurability of these early systems drove many other DL initiatives to develop their own software. Good examples of these are projects such as the *Applied Math and Science Education Repository* (AMSER), *the AAPT ComPADRE Physics and Astronomy Digital Library*, the *Alexandria Digital Library* (ADL) , the (now defunct) *Digital Library for Earth System Education* (DLESE) and quite a few others. Although these projects developed most of their own software, they still used off-the-shelf generic components and generic shell systems. For instance, ComPADRE used ColdFusion, a web application platform for deploying web page components. Likewise, most of these systems rely on database software acquired from database management system makers such as Oracle, IBM, or others, and all of them rely on standard and generic web (HTTP) servers for serving their web pages. As for TE 1.0, it too used quite a few standard, third party components such as the *Apache HTTP web server*, the *XMLFile* program for serving metadata, the *MySQL* relational database, Altova's *XMLSpy* for formulating document standards, the world-wide web consortium's (W3C) HTML and URL (weblink) checkers, and a few more. Still, these earlier DL systems contained a lot of custom-written code.

2. Important note. In *Beyond Software Architecture* Homann (2005) refers to a phenomenon known as résumé-driven design. With this he means that system designers may favor design choices which appeal to them from the perspective of learning or exploring new technologies. Similarly, since system designers and software engineers are in the business of, well..., building (programming) systems, they may be predispositioned to favor building a system over using off-the-shelf solutions. For reasons of full disclosure, both of us were not disappointed in —and had some real influence on— the decision to build rather than to buy.

What About TE 2.0?

In 2015, it was decided that the TE 1.0 architecture, although still running fine, was ready for an overhaul to make it more flexible, have better performance, use newer coding standards, increase security and provide better opportunities for system extensions as well as some end-user modifiability. Between 2002 (TE 1.0 conception) and 2015 (TE 2.0 conception), information technology advanced a great deal. Machines became a lot faster, new programming languages were born and raised, nonrelational databases made a big comeback, virtual machines became commonplace and 'the cloud' came alive in its various forms such as Software as a Service (SaaS) and Platform as a Service (PaaS). Another significant difference between the 2002 and 2015 Web technology landscapes was the new availability of so-called content management frameworks (aka content-management systems or CMS). Although less specific than the aforementioned DL generic systems, these systems are aimed at providing full functionality for servicing content on the web, including user interface configurability. One popular example of such a system is *Drupal*, a community-maintained and open-source CMS which in 2015 was deployed at more than one million websites including the main websites of both our institutions, Oregon State University and the University of Colorado, Boulder.[3] We, therefore, should answer the following question. If at the time of TE 2.0 re-architecting these CMS's were broadly available, why did we decide to once again build TE 2.0 from the ground up rather than implementing it in one of these CMS's?

Indeed, using one of the existing open-source content management systems as a foundation for TE 2.0 would have had a number of advantages. The most popular content management systems: *WordPress*, *Joomla* and *Drupal*, have all been around for quite some time, have active development communities and have rich collections of add-in widgets, APIs and modules for supplementing functionality. In many ways, these content management systems provide turn-key solutions with minimal to no coding required.

Yet, a few concerns drove us towards building TE 2.0 rather than using an existing CMS as its foundation. Out of the box, CMS's are meant to facilitate the publishing of loosely structured content in HTML form. TE curricular documents, on the other hand, are very structured. As discussed in the previous chapter, each document follows one of five prescribed templates containing specific text sections along with metadata such as grade levels, educational standards, required time, estimated cost, group size, *etc*. Curriculum documents are also hierarchically related to each other. For example, *curricular units* can have (child) *lessons*, which in turn can have (child) *activities* (Figure 1).

3. To detect if a website is Drupal based, point your web browser to the website and display the page's HTML source code (right-click somewhere in the page —> *View Page Source*). Drupal pages contain the *generator* meta tag with its *content* attribute set to *Drupal x* where *x* is the Drupal version number.

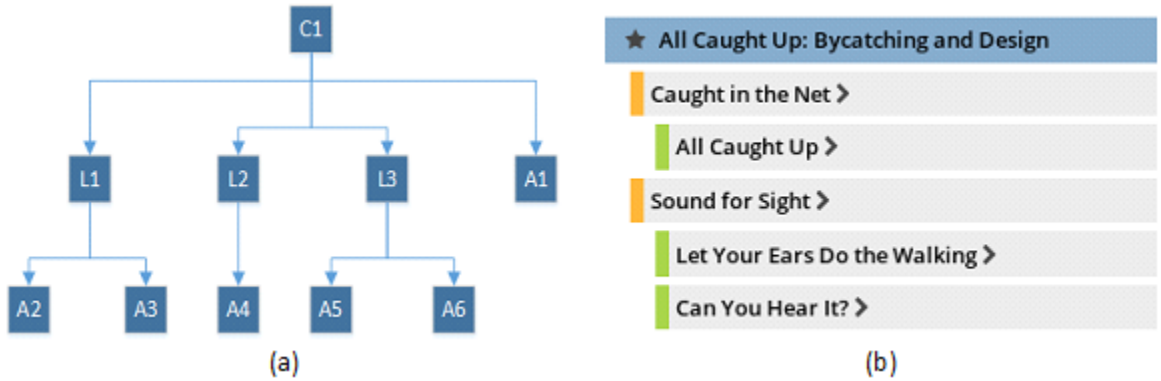


Figure 1: *Hierarchical structure of TE documents. (a) General example. The curricular unit C1 has three lessons (L1-L3) and six activities (A1-A6). Five activities (A2-A6) reside on lessons and one (A1) resides directly on the unit. (b) The All Caught Up curricular unit consists of two lessons and three activities.*

It would of course have been possible to customize any of the popular content management systems to work with this structured data. However, customizing a content management system requires a developer to not only be familiar with the programming languages, databases, and other tools used to build the CMS, but also its Application Programming Interfaces (APIs) and extensibility points. This steepens the learning curve for developers.

Second, since the development of the popular CMS's in the early 2000's, so-called schema-free NoSQL *document databases* have emerged as powerful tools for working with structured and semi-structured data of which TE curricular documents are a good example[4]. CMS support for document databases was limited in 2015. Of the "big three" mentioned earlier, only *Drupal* listed limited support for a single document database (*MongoDB*).

Third, although CMS's increasingly allow developers to give the systems their own look and feel, they do enforce certain conventions and structures. These apply to screen layout and visual components, but also to how the CMS reaches out to external data sources and how end users interact with it. Although this by no means implies that one could not develop a TE-like system within these constraints, being free from them has its advantages. Whereas for content providers with limited coding and development capabilities these constraints represent a price well worth paying in exchange for easy-to-use, predefined layout options, for providers such as the TE team staffed with professional software developers, the reverse might be the case.

Finally, the three most popular content management systems were built using the PHP programming language. TE 2.0 was developed at the Integrated Teaching & Learning Laboratory (ITLL) at the University of Colorado, Boulder. ITLL had two full-time software developers on staff and the other software it develops and supports is primarily based on the Microsoft .Net platform. Given the small size of the development team, there was a strong desire to not introduce another development programming stack to first learn and master, then support and maintain. While there were a few fairly popular content management systems built on the .Net platform, for example, *Orchard* and *DotNetNuke*, they do not have nearly the same level of adoption as the content management systems built on PHP.

4. An important difference between TE 1.0 and TE 2.0 is the switch from a relational (SQL) database backend to a NoSQL backend. We extensively discuss this switch in later chapters

Still despite of TE 2.0 once again having been rebuilt from the ground up, it relies quite heavily on external services, especially when compared with TE 1.0. Table 2.1 contains a side-by-side of a number of functions present in both TE 1.0 and TE 2.0. The ones rendered in green are outsourced to external service providers.

Table 2.1: TE 1.0 *vs.* TE 2.0: on-site *vs.* external services (green fields indicate 'outsourced').

TE 1.0 (2003 – 2016)	TE 2.0 (2016 – present)
On-site, self-managed Apache web server	Azure-managed IIS web server
On-site, self-managed MySQL database	Cloud-managed RavenDB service
Google site search	Azure Search
On-site, self-managed Open Journals System	Cloud-managed Open Journals System service
Achievement Standard Network XML service	Achievement Standard Network JSON service
Google Analytics	Google Analytics
No social networking	Cloud-managed AddThis service
On-site commenting and customer feedback system	Cloud-managed LinkEngineering service Cloud-managed Disqus commenting service
No email marketing	Cloud-managed MailChimp

A Word on Open Source

A question we are sometimes asked is whether we could have (re)built TE open source and whether or not TE is open source?

The first question is easier to answer than the second. No, we do not think that TE, or most systems for that matter, could have initially been developed as open source. The typical open source model is that the

initial developer writes a first, working version of an application, then makes it available under a free or open source software (FOSS) license and invites others to contribute to it. Two classic examples are the GNU software collection and the Linux kernel. When Richard Stallman set out in 1983 to build GNU, a free version of the Unix operating system, he first developed components himself and then invited others to join him in adding components and improving existing ones. Similarly, when Linus Torvalds announced in August 1991 that he was working on the Linux kernel, he had by then completed a set of working components to which others could add and modify. More recent examples are the Drupal CMS mentioned earlier, open sourced by Dries Buytaert in 2001, and .NET Core, Microsoft's open source version of .NET released in 2016. In each of these cases, a set of core functionality was developed prior to open sourcing them.

The straight answer to the second question —is TE open source?— is also 'no,' but only because we estimate that open sourcing is more work than we are willing to take on, not because we do not want to share. It is important to realize that open sourcing a code base involves more than putting it on a web or FTP site along with a licensing statement. One can do that, of course, and it might be picked up by those who want to try or use it. However, accommodating changes, additions, and documentation requires careful management of the code base and this implies work which until now we have hesitated to take on.

Exercise 2.1: Build or Buy? – The $150/Year Case

The following recounts a real-world case of engineering, business and administrative perspectives clashing. It shows how, when making build-or-buy decisions, the world can look rather different depending on the perspective through which that world is seen.**The Players**

- Project Lead (PL)
- External Software Development Team (SDT)
- Infrastructure / systems supervisor and assistant (IT)
- Cloud Services Provider (CSP)

The Context

A $3M, 3-year project overseen and lead by the PL. One of the components of the project is the development of a data visualization application. The development of the visualization application has been outsourced to an external SDT at a cost of $165K. Once the development is complete, the application must be hosted on the systems of the organization running the project.

The Problem — Part 1

The software development team has completed the development and testing of the data visualization app. The application is computationally intensive (requires lots of CPU cycles). They have developed it on a VM running on the systems of their own organization. The VM was set up as having two (2) single-core CPUs and 16 GB of memory. The application seems to run fine, with good performance, even on the largest of test cases. The SDT has not spent all of its budget; about 25% of the budget is remaining. The SDT has lots of ideas for additional functionality and collection of usage data.

The project's organization does not host any of its applications on its own computers; virtualized or otherwise. Instead, they deploy all their applications using an external CSP. Following that model, IT sets up a VM at the CSP to run the visualization app. The VM is of the least expensive option: one (1) single-core CPU and 1.75 GB of memory. Cost per month: $12.50.

As soon as the system is running, the SDT runs some tests and notices that the application performs very poorly on larger, more computationally intensive test cases. In fact, they discover that while these cases run, all other requests coming in for that same service —even the ones which would not require much memory or CPU time— are queued up and have to wait until the original, large test case has completed. Since actual users will expect a result within a few seconds, the SDT considers this unacceptable and contacts IT to see what can be done about this.

After running its own tests, IT assesses that the problem almost certainly lies with the limited capacity of the VM. They suggest that the SDT work around the problem by modifying the application so that it pre-caches computationally expensive cases. In other words, they suggest additional development of a module that anticipates those large cases, precomputes their solutions during times of low demand, stores (caches) those solutions and then simply returns those (precomputed and cached) solutions whenever they are requested, rather than computing them on the fly.

Whereas the SDT considers this an intriguing suggestion, they react not by embarking on this caching approach, but instead by asking the project organization what it would cost to upgrade the VM. IT answers that the CSP offers doubling the VM's capacity —two (2) single-core CPUs and 3.5 GB of memory— at $25 per month. However, they do not offer to buy that upgrade. Instead, they insist on the SDT developing the caching module.

Question Set 1 (please think about/answer this before(!!) you move on to the second part of this problem.

- Why did the SDT ask for upgrading the VM rather than embarking on the development of a caching module? When thinking about this, consider that the SDT pays its (student) developers $18/hr. Hint: it is not just development cost that the SDT has in mind.
- Why would IT not offer to buy the VM upgrade, but instead insist on the SDT developing the caching module?
- Why might the project organization not 'spin up' its own VM and host the application on it, especially since it has plenty of computing capacity?
- If we assume that both the project organization and the SDT want the application to be a success, what course of action do you recommend at this point?

The Problem — Part 2

After some back-and-forth, IT decide to buy the VM upgrade at a cost of $25/month ($300/year). They also note that the CSP offers doubling the VM capacity once again at a doubling of the price ($600/year) but state that it will not do that without explicit permission of the PL. The SDT thanks IT for the VM upgrade.

Question 2

What should be done if the initial upgrade does not prove sufficient and large cases continue to run slow? Think of some options and consider the pros and cons of those options for each of the parties involved.

References

Ford, P. (2015) The Code Issue. Special Issue on Programming/Coding. *Bloomberg Business Week Magazine*. June 2015.

Homann, L. (2005) *Beyond Software Architecture*. Addison-Wesley.

Sullivan, J. J., Beach, R. (2004). A Conceptual Model for Systems Development and Operation in High Reliability Organizations. In: Hunter, M. G., Dhanda, K. (Eds.). *Information Systems: Exploring Applications in Business and Government*. The Information Institute. Las Vegas, NV.

3. TE 1.0 – XML

Introduction

TE 1.0 relied heavily on Extensible Markup Language (XML). XML was invented in the second half of the 1990s to overcome a fundamental problem of the early world-wide web. The problem was that the predominant language for representing web content, HyperText Markup Language (HTML), was meant to express how information was to be formatted on web pages to be viewed by human users, but that that formatting was of little use to 'users' represented by machines; *i.e.*, programs. Whereas humans are quite good at extracting meaning from how information is formatted, programs just need content, and formatting only gets in the way of extracting that content. Yet HTML was meant to specify content through formatting. XML solved this problem by providing a text-based and structured way to specify content without formatting.

In this chapter, we introduce XML as a data representation and data exchange format and provide some examples of how it was used in TE 1.0. In the next chapter, we discuss XML's recent competitor and TE 2.0's choice: JSON. In the chapter following the JSON chapter we go deeper into how XML was used in TE 1.0 and how JSON is used in TE 2.0.

Representing Content With XML

One of the more influential advances in modern-day electronic data exchange, and one which caused web-based data exchanges to flourish, was the introduction and standardization of Extensible Markup Language (XML) in the late 1990s (Bosak & Bray, 1999). Until that time, messages requested and served over the web were dominated by the HyperText Markup Language (HTML). As explained in the 1999 article in *Scientific American* by Jon Bosak and Tim Bray —two of the originators of the XML specification— HTML is a language for specifying how documents must be formatted and rendered, typically so that humans can easily read them. However, HTML is not very well suited for communicating the actual content of documents (in terms of their information content) or for that matter, any set of data. This deceptively simple statement requires some explanation.

When we, as humans, inspect web pages, we use their formatting and layout to guide us through their organization and contents. We look at a page and we may see sections and paragraphs, lists and sublists, tables, figures, and text, all of which help us to order, structure, and understand the contents of the document. In addition, we read the symbols, words, figure captions, and sentences, gleaning their semantic contents from the terms and expressions they contain. As a consequence, on a web page we can immediately recognize the stock quote or the trajectory of the share price over the last six hours from a chart, a table or even a text. Since a human designed the page to be read and processed by another human, we can count on each other's perceptual pattern recognition and semantic capabilities when exchanging information through a formatted text. We are made painfully aware of this when confronted with a badly structured, cluttered or poorly formatted HTML web page or when the page was created by someone with a different

frame of mind or a different sense of layout or aesthetics. What were the authors thinking when they put this page together?

However, if we want to offer contents across the web that must be consumed by programs rather than human beings, we can no longer rely on the formats, typesetting and even the terms of the document to implicitly communicate meaning. Instead, we must provide an explicit semantic model of the content of the document along with the document itself. It is this ability to provide content along with a semantic model of that content that makes XML such a nice language for programmatic data exchange.

Although for details on XML, its history, use and governance, we refer to the available literature on this topic; we provide here a small example of this dual provision of contents and semantics.

Consider TeachEngineering: an electronic collection of lesson materials for K-12 STEM education. Now suppose that we want to give others; *i.e.*, machines other than our own, access to those materials so that those machines can extract information from them. What should these lesson materials look like when requested by such an external machine or program? Let us simplify matters a little and assume that a TE lesson consists of only the following:

- Declaration that says it is a lesson
- Lesson title
- Target grade band
- Target lesson duration
- Needed supplies and their estimated cost
- Summary
- Keywords
- Educational standards to which the lesson is aligned
- Main lesson contents
- References (if applicable)
- Copyright

In XML such a lesson might be represented as follows:

```
<lesson>
  <title>Hindsight is 20/20</title>
  <grade target="5" lowerbound="3" upperbound="6"/>
  <time total="50" unit="minutes"/>
  <lesson_cost amount="0" unit="USDollars"/>
  <summary>Students measure their eyesight and learn how lenses can
  enhance eyesight.
  </summary>

  <keywords>
    <keyword>eyesight</keyword>
    <keyword>vision</keyword>
    <keyword>20/20</keyword>
  </keywords>
```

```
  <edu_standards>
    <edu_standard identifier="14000"/>
    <edu_standard identifier="14011"/>
  </edu_standards>

  <lesson_body>With our eyes we see the world around us. Having two
  eyes helps us see a larger area than just one eye and with two
  eyes we can... etc. etc.
  </lesson_body>
  <copyright owner="We, the legal owners of this document"
   year="2016"/>
</lesson>
```

Notice how the various components of a lesson are each contained in special tags such as `<copyright>` or `<title>`. Hence, to find which educational standards this lesson supports, all we have to do is find the `<edu_standards>` tag and each of the `<edu_standard>` tags nested within it.

Exercise 3.1

Copy the above XML fragment to a file called *something.xml* and pick it up with your web browser (*Control-O* makes your web browser pop up a file browser). Notice that your web browser recognizes the content of the file as XML and renders it accordingly. Note how the indentation of the various lines matches the nesting of the data. For instance, your browser indents `<keyword>`s because they are contained within the `<keywords>` tag. The same applies to the `<edu_standards>` and `<edu_standard>` tags.

The above XML format is rather rigid and not entirely pleasant for us humans to read. However, it is this rigid notion of information items placed inside these tags and tags themselves placed within other tags that provides three essential advantages for machine-based reading:

1. Information is represented in a hierarchical format; *i.e.*, tags inside other tags. Hierarchies provide a lot of expressiveness; *i.e.*, most (although not all) types of information can be expressed by means of a hierarchy.
2. It is relatively easy to write programs that can read and process hierarchically organized data.
3. If such a program cannot successfully read such a data set, it is likely that the data set is not well formed and hence, we have a good means of distinguishing well-formed from malformed data sets.

Let us take a look at another example. Figure 1 contains the music notation of a fragment from Beethoven's famous fifth symphony (listen to it!):

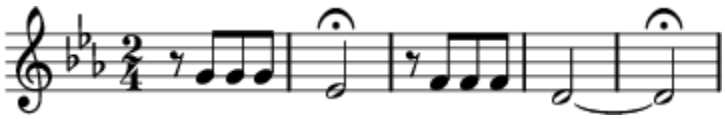
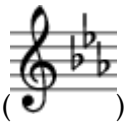
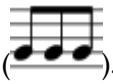
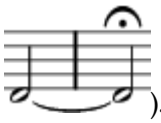


Figure 1: *The first five bars of the main melody of Beethoven's Symphony No 5.*

For those who can read music notation, the information contained in this image is clear:

- The piece is set in the key of C-minor () (or E-flat major: from these first few bars you cannot really tell which of the two it is). This implies that E's, B's and A's must be flattened ().
- Time signature is 2/4; *i.e.*, two quarter-note beats per bar ().
- First bar consists of a ½-beat rest () followed by three ½-beat G's ().
- Second bar contains a 2-beat E-flat (). The note has a *fermata* (), indicating that the performer or conductor is free to hold the note as long as desired.
- The two 2-beat D's in bars four and five must be connected (tied or 'slurred') and played as a single 4-beat note. Once again, this note has a *fermata* and can therefore be held for as long as desired ().

Fot those who can read music, all this information is stored in the graphic patterns of music notation. For instance, it is the location of a note relative to the five lines in the staff and the staff's *clef* () which indicates its pitch, and the duration of a note is indicated by whether it is solid or open or how notes are graphically connected. Likewise, the *fermata* is just a graphical symbol which we interpret as an instruction on how long to hold the note.

However, whereas this kind of information is relatively easy for us humans to glean from the graphic patterns, for a machine, this is not nearly so easy. Although in these days of optical pattern recognition and machine learning we are quickly getting closer to this, a much more practical approach would be to write this same information in unformatted text such as XML so that a program can read it and do something with it.

What might Beethoven's first bars of Figure 1 look like in XML? How about something like the following?[1]

```
<score>
  <clef>g</clef>
  <key base_note="c" qualifier="minor">
    <key_modifiers>
      <note>E</note><mod>flat</mod>
      <note>B</note><mod>flat</mod>
      <note>A</note><mod>flat</mod>
```

1. Please note: the XML offered here is *a l'improviste* (off the cuff) and not at all based on a good model of music structure. It is merely meant to provide an example of how, in principle, a music score can be represented in XML.

```
    </key_modifiers
  </key>
  <time_signature numerator = "2" denominator="4"/>
  <bars>
    <bar count="1">
      <rest bar_count = "1" duration="8"/>
      <note bar_count = "2" pitch="G" duration="8"/>
      <note bar_count = "3" pitch="G" duration="8"/>
      <note bar_count = "4" pitch="G" duration="8"/>
    </bar>
    <bar count="2">
      <note bar_count = "1" pitch="E" duration="2"
      articulation="fermata"/>
    </bar>
    .
    .
    .
    Etc.
  </bars>
</score>
```

Looking at the example above, you may wonder about why some information is coded as so-called XML *elements* (entries of the form `<tag>information</tag>`), whereas other information is coded in the form of so-called *attributes* (entries in the form of `attribute="value"`). For instance, instead of

```
<note bar_count = "4" pitch="G" duration="8"/>
```

could we not just as well have written the following?

```
<note>
  <bar_count>4</bar_count>
  <pitch>G</pitch>
  <duration>8</duration>
</note>
```

The answer is that either way of writing this information works just fine. As far as we know, a choice of one or the other is essentially a matter of convenience and aesthetics on the side of the designer of the XML specification. As we mention in the next chapter (JSON; an Alternative for XML), however, this ambivalence is one of the arguments that JSON aficionados routinely use against XML.

At this point you should not be surprised that indeed there are several XML models for representing music. One of them is MusicXML.[2]

2. Several competing specifications for Music XML exist. In this text we care only to convey the notion of writing an XML representation for sheet/score music and take no position on which XML format is to be preferred.

This notion of communicating information with hierarchically organized text instead of graphics naturally applies to other domains as well. Take, for instance mathematical notation. Figure 2 shows the formula for the (uncorrected) standard deviation.

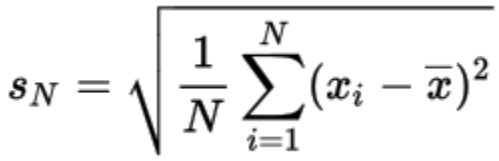


Figure 2: *Formula for the (uncorrected) standard deviation.*

Once again, we humans can read this information just fine because we have a great capacity (and training) to glean its meaning from its symbols and their relative positions. But as was the case with the music graphic, asking a machine to decipher these same symbols and relative positions seems like the long way around. Why not just represent the same information in hierarchic textual, XML form? Something like the following:

```
<equation>
  <left_side>
    <term>
      <symbol>S</symbol>
      <sub><var>N</var></sub>
    </term>
  </left_side>
  <right_side>
    <sqrt>
      <product>
        <left>
          <quotient>
            <nominator>1</nominator>
            <denominator><var>N</var></denominator>
          </quotient>
        </left>
        <right>
          <sum>
            Etc.
          </sum>
        </right>
      </product>
    </sqrt>
  </right_side>
</equation>
```

As with the music example, this XML was entirely made up by us and is only meant to illustrate the notion of representing content in hierarchical text form which is normally represented in graphic form. However, MathML is a standard implementation of this.

What is interesting in MathML is that it consists of two parts: *Presentation MathML* and *Content MathML*. Whereas *Presentation MathML* is comparable with standard HTML, *i.e.*, a language for specifying how mathematical expressions must be displayed, *Content MathML* corresponds to what we tried to show above, namely a textual presentation of the structure and meaning of mathematical expressions. The following sample is taken verbatim from the MathML Wikipedia page:

Expression: $ax^2 + bx + c$

MathML:

```
<math>
  <apply>
    <plus/>
    <apply>
      <times/>
      <ci>a</ci>
      <apply>
        <power/>
        <ci>x</ci>
        <cn>2</cn>
      </apply>
    </apply>
    <apply>
      <times/>
      <ci>b</ci>
      <ci>x</ci>
    </apply>
    <ci>c</ci>
  </apply>
</math>
```

Lots of XML specifications other than MusicML and MathML have been developed over the years. One which is currently in active use by the US Securities and Exchange Commission (SEC) is XBRL for business reporting (Baldwin & Brown, 2006). For a list of many more, point your web browser to https://en.wikipedia.org/wiki/List_of_XML_markup_languages.

XML Syntax Specification: DTD and XML Schema

One of the characteristics of programs which serve XML content over the web, is that they can be self-describing using a so-called Document Type Definition (DTD) or the more recent XML Schema Definition (XSD). DTS and XSD are meta documents, meaning that they contain information about the documents

containing the actual XML data. This meta information serves two purposes: it informs programmers (as well as programs) on how to interpret an XML document and it can be used to check an XML document against the rules specified for that XML (a process known as 'validation').

A helpful way to understand this notion is to consider a DTD/XSD document to specify the syntax —grammar and vocabulary— of an XML specification. For instance, going back to the example of our TeachEngineering 1.0 lesson, the DTD/XSD for the lesson would specify that a lesson document must have a title, a target grade band, one or more keywords, one or more standard alignments, time and cost estimates, *etc*. It would further specify that a grade band contains a target grade and a low and a high grade which are numbers, that the keyword list contains at least one keyword which itself is a string of characters, that a copyright consists of an owner and a year, *etc*.

A fragment of the XSD for the above lesson document defining the syntax for the grade, time and keyword information might look something like the following (Note: line numbers are included here for reference only; they would not be part of the schema document):

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3c.org/2001/XMLSchema">
3 <xs:element name="lesson">
4   <xs:complexType>
5    <xs:sequence>
6
7       <!--semantics for the <grade> element-->
8       <xs:element name="grade">
9         <xs:complexType>
10           <xs:simpleContent>
11            <xs:extension base="xs:unsignedByte">
12              <xs:attribute name="lowerBound" type="xs:unsignedByte"
13                use="optional"/>
14              <xs:attribute name="upperBound" type="xs:unsignedByte"
15                use="optional"/>
16            </xs:extension>
17          </xs:simpleContent>
18        </xs:complexType>
19      </xs:element>
20
21      <!--semantics for the <time> element-->
22      <xs:element name=”time”>
23        <xs:complexType>
24          <xs:simpleContent>
25            <xs:extension base="xs:float">
26              <xs:attribute name="unit" use="required">
27                <xs:simpleType>
28                  <xs:restriction base="xs:string">
29                    <xs:enumeration value="minutes"/>
```

```
30                    <xs:enumeration value="hours"/>
31                    <xs:enumeration value="days"/>
32                    <xs:enumeration value="weeks"/>
33                  </xs:restriction>
34                </xs:simpleType>
35              </xs:attribute>
36            </xs:extension>
37          </xs:simpleContent>
38        </xs:complexType>
39      </xs:element>
40
41      <!--semantics for the <keywords> element-->
42      <xs:element name="keywords">
43        <xs:complexType>
44          <xs:sequence>
45            <xs:element name="keyword" type="string" maxOccurs="unbounded"/>
46          </xs:sequence>
47        </xs:complexType>
48      </xs:element>
49  Etc.
```

Notice how the schema specifies how components of an XML lesson must be structured. For instance, the `<keywords>` element (line 42-48) is defined as a sequence of `<keyword>`s where each `<keyword>` is a string of characters of any length (line 45). Similarly, the lesson `<time>` (line 22-39) has a value which is a floating-point number and has a required `<unit>` which is one of the strings' minutes, hours, days or weeks (line 28-33).

As for the *xs:* prefix on all definitions, the code

```
xmlns:xs="http://www.w3c.org/2001/XMLSchema"
```

on line 2 indicates that each of these terms —*element*, *complexType*, *sequence*, *string*, *etc.*— is defined by the W3C's 2001 *XMLSchema*.

Note: Well formed ≠ Valid

Now we have discussed both XML and DTD/XSD, we can make the distinction between well-formed and malformed XML documents on the one hand and valid and invalid ones on the other (Table 1).

Table 1: relationships between XML well-formedness and validity

	Well formed	Malformed
Valid	1	
Invalid	2	3

An XML document is considered *well formed* if it obeys the basic XML syntax rules. With this, we mean that all content is stored within XML elements; *i.e.*, that all content is tag delimited and properly nested. A simple example/exercise clarifies this.

Exercise 3.2

Store the following text in a file with *.xml* extension and pick it up with your web browser (*Control-o* makes the web browser pop up a file browser):

```
<art_collection>
 <object type="painting"
   <title>Memory of the Garden at Etten</title>
   <artist>Vincent van Gogh</artist>
   <year>1888</year>
   <description>Two women on the left. A third works in her
   garden</description>
   <location>
     <place>Hermitage</place>
     <city>St. Petersburg</city>
     <country>Russia</country>
   </location>
 </object

 <object type="painting">
   <title>The Swing</title>
   <artist>Pierre Auguste Renoir</artist>
   <year>1886</year>
   <description>Woman on a swing. Two men and a toddler
   watch</description>
   <location>
     <place>Musee d'Orsay</place>
     <city>Paris&lt;/city>
     <country>France</country>
   </location>
```

```
  </object>
</art_collection>
```

Notice how your web browser complains about a problem on line 3 at position 9. It sees the `<title>` element, but the previous element `<object>` has no closing chevron (>) and hence, the `<title>` tag is in an illegitimate position. Regardless of any of the values and data stored in any of the elements, this type of error violates the basic syntax rules of XML. The document is therefore not well formed and any malformed document is considered invalid (cell 3 in Table 1).

However, an XML document can be well formed yet still be invalid (cell 2 in Table 1). This occurs if the document obeys the basic XML syntax rules but violates the rules of the DTD/XSD. An example would be a well-formed TeachEngineering lesson which does not have a summary or a grade specification. Such an omission does render the document invalid, even though it is well formed.

Must All XML Have a DTD/XSD?

A question which often pops up when discussing XML and DTD/XSD is whether or not DTD/XSD are mandatory? Must we always go through the trouble of specifying a DTD/XSD in order to use XML? A related question is whether we must process a DTD/XSD to read and consume XML? The answer to both questions is 'no,' a DTD/XSL is not required and even if it has a DTD/XSD, there is no requirement to process that DTYD/XSD. However, if, as a data provider you make XML available to others, it is good practice to create and publish a DTD or XSD for it so that you can validate your XML before you expose it to the world. If, on the other hand, you are a consumer of XML, it is often sufficient to just read the on-line documentation of the XML web service you are consuming, and you certainly do not have to generate a DTD/XSD yourself.

Enough Theory. Time For Some Hands-on

Exercise 3.3

Point your browser to *https://classes.business.oregonstate.edu/reitsma/family.xml*. Notice that we have a small XML data set of a family of two people:

```
  <?xml version="1.0" encoding="UTF-8"?>
<family>
  <person gender="male">
```

```
    <firstname>Don</firstname>
    <lastname>Hurst</lastname>
  </person>
  <person gender="female">
    <firstname>Mary</firstname>
    <lastname>Hurst</lastname>
  </person>
</family>
```

Let us now write a Python (3.x) program which can pick up (download) this data set and extract the information from it. (Make sure that you understand the larger picture here. Assume that instead of having a hardwired, static XML data set on the web site, we can pass the web service a family identifier and it will respond, on-the-fly, with a list of family members in XML. For instance, we might send it a request asking for the members of the *Hurst* family upon which it replies with the data above).

Please note that there exists a variety of ways and models to extract data from XML files or web services. Suffice it here to say that in the following Python (3.x) and JavaScript examples the XML is stored in memory as a so-called *Document Object Model (DOM)*. A DOM is essentially a hierarchical, tree-like memory structure (we have already discussed how XML documents are hierarchical and, hence, they nicely fit a tree structure). Again, several methods for extracting information from such a DOM tree exist.

Store the following in a file called *family.py* and run it.

```
  import requests
import xml.etree.ElementTree as ET

url = "https://classes.business.oregonstate.edu/reitsma/family.xml"

#Retrieve the XML over HTTP
try:
  response = requests.get(url)
except Exception as err:
  print("Error retrieving XML...\n\n", err)
  exit(1)

#Build the element tree
try:
  #Read and parse the XML
  root = ET.fromstring(response.text)
except Exception as err:
  print("Error parsing XML...\n\n", err)
  exit(1)
```

```
#Name (tag) of root element
print("Name (tag) of root element: " + root.tag)

#Child elements of root
print("\nChild elements:")
for child in root:
  print(child.tag, child.attrib)

#Access the children and grandchildren by index
print("\nChild element content by index:")
print(root[0][0].text + " " + root[0][1].text)
print(root[1][0].text + " " + root[1][1].text)

#Loop over the children and get their children
print("\nChild element content by element:")
for child in root.findall("person"):
  print(f"{child[0].text} {child[1].text}: {child.attrib['gender']}")
```

Output:

```
Name (tag) of root element: family

Child elements:
person {'gender': 'male'}
person {'gender': 'female'}

Child element content by index:
Don Hurst
Mary Hurst

Child element content by element:
Don Hurst: male
Mary Hurst: female
```

Output:

```
Name (tag) of root element: family

Child elements:
person {'gender': 'male'}
person {'gender': 'female'}
```

```
Child element content by index:
Don Hurst
Mary Hurst

Child element content by element:
Don Hurst: male
Mary Hurst: female
```

Exercise 3.4

One of the advantages of JavaScript is that pretty much all web browsers have a JavaScript interpreter built in. Hence, there is nothing needed beyond your web browser to write and run JavaScript code. Here is the JavaScript program. Note that the JavaScript is embedded in a little bit of HTML (the JavaScript is the code within the `<script>` and `</script>` tags):

```
  <html>
<p id="family"></p>

<script>
var request = new XMLHttpRequest();
request.onreadystatechange = extract;
request.open("GET",
  "https://classes.business.oregonstate.edu/reitsma/family.xml",
  true);
request.send();

function extract()
{
  if (request.readyState == 4 && request.status == 200)
  {
    var xmlDoc = request.responseXML;
    var my_str = "";
    var persons = xmlDoc.getElementsByTagName("person");
    for (var i = 0; i < persons.length; i++)
    {
      var firstname = persons[i].childNodes[1];
      var lastname = persons[i].childNodes[3];
```

```
      my_str = my_str + firstname.innerHTML + " "
      + lastname.innerHTML + "<br/>";
    }
    document.getElementById("family").innerHTML = my_str;
  }
}
</script>
</html>
```

You may try storing this file with the **.html* extension on your local file system and then pick it up with your browser, but that will almost certainly not work because this implies a security risk.[3] To make this work, however, we installed the exact same code at *https://classes.business.oregonstate.edu/reitsma/family.html* and if you point your browser there, things should work just fine. (To see the HTML/JavaScript source code, *right-click View Page Source* or point your browser to *view-source:https://classes.business.oregonstate.edu/reitsma/family.html*)

TE 1.0 Documents Coded and Stored as XML

In their 1999 article in Scientific American, Bosak and Bray considered an XML-equipped web the "*Next Generation Web*". With this, they meant that until that time, Web content carried in HTTP was meant to be presented to humans, whereas with XML we now had a way to present content format-free to machines. Moreover, along with XML came tools and protocols which made it relatively easy to programmatically process XML.

When we developed TE 1.0 in the early 2000s, we liked this concept so much that we decided to store all TE content in XML.

Exercise 3.5

To see this, first take a look at an arbitrary TE 1.0 activity at:

3. The security risk rests in the web browser running JavaScript code instructing it to reach out to an external server. This interacting of an HTTP client (the web browser) with an external source to dynamically generate content is known as *Asynchronous JavaScript with* XML (AJAX) technology. By default, reaching out to a server located in another domain than the one from which the original content was requested—aka *Cross Origin Resource Sharing* (CORS) —is forbidden (one can easily imagine typing in a login and password which is then used to access a third party site; all in real-time from the user's computer, yet entirely unbeknownst to the user).

https://cob-te-web.business.oregonstate.edu/view_activity.php?url=collection/mis_/activities/mis_eyes/mis_eyes_lesson01_activity1.xml

When you look at the page source in your browser (right-click: *View page source*), you immediately see that the page is an HTML page (the first tag is the <*html*> tag). This makes much sense, as the page is meant to be read by human users and hence, HTML is a good way of rendering this information in a web browser.

However, a quick look at the activity's URL shows that the program *view_activity.php* is passed the parameter *url* which is set to the value *collection/mis_/activities/mis_eyes/mis_eyes_lesson01_activity1.xml*.

If we thus point our browser to: *https://cob-te-web.business.oregonstate.edu/collection/mis_/activities/mis_eyes/mis_eyes_lesson01_activity1.xml*, we see the actual XML holding the activity information. Each activity has 14 components: `<title>`, `<header>`, `<dependency>`, `<time>`, `<activity_groupsize>`, *etc.* Most of these are complex types in that they have one or more components of their own.

When TE 1.0 receives a request to render an activity, its *view_activity.php* program extracts the various components from the associated XML file, restrtuctures/formats them into HTML and serves the HTML to the requester.

view_activity.php does some other things as well. For instance, scroll down through the activity XML to the `<edu_standards>` tag. Notice how for this activity seven standards are defined, each with an *Sxxxxxxx* identifier (`<edu _standard identifier="Sxxxxxxx">`). However, when you switch back to the HTML view and look for the *Educational Standards* section, you find the text of those standards rather than their identifiers. How's that done? Quite simply, really. As *view_activity.php* renders the activity, it extracts the `<edu _standard>`tags and their identifiers from the XML. It then queries a database which holds these standards with those identifiers for the associated standard texts, their grade level, their geographic origin, *etc.* Having received this information back from the database, it encodes it in HTML and serves it up as part of the activity's HTML representation.

Service-Oriented Architectures and Business Process Management

Reading 'between the lines,' one can see a grander plan for how whole system architectures based on XML (or JSON) web services can be built: a company-wide or world-wide network of information processing software services that is utilized by software programs that connect to this network and request services from it. Service requesters and providers communicate with each other through common protocols and expose each other's interfaces through which they exchange information (Berners-Lee *et al.*, 2001).

One expression of this vision is what are known as *Service-Oriented Architectures* (SOA). MacKenzie *et al.* (2006) define SOA as a "*paradigm for utilizing and organizing distributed capabilities that may be under the control of different ownership domains*;" in other words, the distribution of software components over machines, networks and possibly organizations that are accessed as web services. Whereas in a traditional system architecture we would embed functionality within the applications that need it, in an SOA, our application software would fulfill the role of a communications officer and information integrator with most, if not all of the functionality provided by (web) services elsewhere on the network. Some of these services

might run on our own machines; others can reside at third parties. Some may be freely available whereas others may be 'for fee.'

Regardless of where these services reside and who owns them, however, they all can be accessed using some or all of the methods that we have discussed above. They exchange information in forms such as XML, they receive and send messages with protocols such as SOAP and they are self describing through the exposure of their XSD, DTD or WSDL (SOAP and WSDL are XML specifications for generalized message exchange). Hence, as long as the applications requesting their information can formulate their requests following these protocols, they can interoperate with the services.

Special update. In 2018 —the year we published the first edition of this text— the term 'web services' referred to XML (or JSON) data exchange services as discussed here. Nowadays —2022— the term 'web service' has been mostly replaced by 'web endpoint,' or 'web API.' There also is a lot of talk about so-called 'micro services.' The idea of micro services is very(!) similar to the just mentioned idea of building larger systems from small, independently functioning components that can communicate information/data with each other. The difference with 'web services' is that 'micro services' is a more general term in that it does not limit the concept to web (HTTP-based) services.

TE 1.0 Web Services Example I: K-12 Standards

As mentioned in the introductory chapter, all of TeachEngineering's curriculum is aligned with K-12 STEM standards. Although the TE team is responsible for these alignments, it is not in the business of tracking the standards themselves. With each of the US states changing its standards, on average, once every five years and with a current total of about several 10,000s such standards, tracking the standards themselves was deemed better to be left to a third party. This party, as previously mentioned, is the *Achievement Standard Network (ASN)* project, owned and operated by the *Desire2Learn* (*D2L*) company.

Very much in the spirit of web services as discussed here, ASN makes its standard set available as an XML-based service.[4]

Here is a (simplified) fragment of one of ASN's standard sets, namely the 2015 South Dakota Science standards. The fragment contains two Kindergarten (K)-level standards:

```
<rdf:RDF xmlns:asn="https://asn.desire2learn.com/resources/S2627378httphttps://purl.org/ASN/s
xmlns:cc="https://creativecommons.org/ns#"
xmlns:dc="https://purl.org/dc/elements/1.1/"
xmlns:dcterms="https://purl.org/dc/terms/"
xmlns:foaf="https://xmlns.com/foaf/0.1/"
xmlns:gemq="https://purl.org/gem/qualifiers/"
xmlns:loc="https://www.loc.gov/loc.terms/relators/"
xmlns:owl="https://www.w3.org/2002/07/owl#"
xmlns:skos="https://www.w3.org/2004/02/skos/core#"
```

4. Although ASN still serves standards as XML if so desired, it has recently shifted to serving them in JSON, a data exchange standard we will discuss and practice in the next chapter.

```
xmlns:rdf="https://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="https://www.w3.org/2000/01/rdf-schema#">

<cc:attributionURL
rdf:resource="https://asn.desire2learn.com/resources/D2627218" />
<dc:title xml:lang="en-US">South Dakota Science
Standards</dc:title>
<dcterms:description xml:lang="en-US">The South Dakota
Science Standards realize a vision for science education
in which students are expected to actively engage in science
and engineering practices and apply crosscutting concepts
to deepen their understanding of core ideas. These standards
are designed to guide the planning of instruction and the
development of assessments of learning from kindergarten
through twelfth grade. This document presents a starting
point for informed dialogue among those dedicated and
committed to quality education in South Dakota. By providing
a common set of expectations for all students in all schools,
this dialogue will be strengthened and enhanced.
</dcterms:description>
<asn:repositoryDate
rdf:datatype="https://purl.org/dc/terms/W3CDTF">
2015-05-19</asn:repositoryDate>

<asn:Statement
rdf:about="https://asn.desire2learn.com/resources/S2627378">
  <asn:statementNotation>K-PS2-1</asn:statementNotation>
  <dcterms:educationLevel
  rdf:resource="https://purl.org/ASN/scheme/ASNEducationLevel/K" />
  <dcterms:subject
  rdf:resource="https://purl.org/ASN/scheme/ASNTopic/science" />
  <dcterms:description xml:lang="en-US">Plan and carry out an
  investigation to compare the effects of different strengths
  or different directions of pushes and pulls on the motion
  of an object.</dcterms:description>
</asn:Statement>

<asn:Statement
rdf:about="https://asn.desire2learn.com/resources/S2627379">
  <asn:statementNotation>K-PS2-1</asn:statementNotation>
  <dcterms:educationLevel
  rdf:resource="https://purl.org/ASN/scheme/ASNEducationLevel/K" />
  <dcterms:subject
```

```
  rdf:resource="https://purl.org/ASN/scheme/ASNTopic/science" />
  <dcterms:description xml:lang="en-US">Analyze data to
  determine if a design solution works as intended to change
  the speed or direction of an object with a push or a pull.
  </dcterms:description>
</asn:Statement>
```

Notice how at the very top, the XML fragment contains a reference to the XSD that governs it (feel free to pull it up in your web browaer):

`https://purl.org/ASN/schema/core/`

Notice also that depending on how much we need to know about this web service, we might or might not need to analyze this XSD. If all we want to do is write a program which grabs the texts of the various standards, we do not really have to know the XSD at all. All we have to know is how to extract the `<dcterms:description>` elements, something which a quick study of the example shows us.

Exercise 3.6
Note how each of these two standard representations contains a link to a more human-readable representation:

`https://asn.desire2learn.com/resources/Sxxxxxxx`

Point your browser to each of these to see what else they hold.

TE 1.0 Web Services Example II: Metadata Provisioning

A second example of the application of web services in TE 1.0 is comprised of metadata provisioning. One of the goals of the National Science Digital Library (NSDL) project mentioned in the introductory chapter was that as a centralized registry of digital science libraries, NSDL would be up-to-date on all the holdings of all of its member libraries. For users —or *patrons* in library jargon— this would mean that they could come to NSDL and conduct targeted searches over all its member libraries without having to separately search these libraries. With '*targeted search*' we mean a search which is qualified by certain constraints. For instance, a South Dakota seventh grade science teacher might ask NSDL if any of its member libraries contains curriculum which supports standard MS-ESS3-1 (*Construct a scientific explanation based on evidence for how the uneven distributions of Earth's mineral, energy, and groundwater resources are the result of past and current geoscience processes*) or for a 10^{th} grade teacher, if there is curriculum which addresses plate tectonics which can be completed within two hours.

Two standard approaches to serve such a query come to mind. The first is known as *federated search*.

In this approach a search query is distributed over the various members of the federation; in this case the various NSDL member libraries. Each of these members would conduct its own search and report back to the central agency (NSDL) which then comprises and sorts the results, and hands them to the original requester.

A second approach is that of the *data hub* in which the searchable data are centrally collected, independent of any future searches. Once a search request comes in, it can be served from the central location without involvement of the individual members.

Exercise: Compare and contrast the federated and data hub approaches to search. What are the advantages and disadvantages of each?

Since the architects of NSDL realized that it was not very likely that all its member libraries would have their search facilities up and running all the time and that they would all function sufficiently fast to support federated searches, it decided in favor of the data hub approach. This implied that it would periodically ask its member libraries for information about their holdings and centrally store this information, so that it could serve searches from it when requested. This approach, however, brings up three questions: 1. what information should the member libraries submit; 2. what form should the information be in; and 3. what sort of data exchange mechanism should be used to collect it? Having worked your way through this chapter up to this point, can you guess the answer to these questions?

- **Question**: What information should the member libraries submit? **Possible answer**: NSDL and its members should collectively decide on a standard set of data items representing member holdings.
- **Question**: What form should the information be in? **Possible answer**: XML is a good candidate. Supported by a DTD/XSD which represents the required and optional data items, XML provides a formalization which can be easily served by the members and consumed by the central entity.
- **Question**: What sort of data exchange mechanism should be used to collect it? **Possible answer**: An HTTP/XML web service should work fine.

Fortunately for NSDL and its member libraries, the second and third questions had already been addressed by the digital library world at large and its Open Archives Initiative (OAI). In 2002, OAI released its Protocol for Metadata Harvesting (OAI-PMH); an XML-over-HTTP protocol for exposing and harvesting library metadata. Consequently, NSDL asked all its member libraries to expose the data about their holdings using this protocol.

Exercise 3.7

To see OAI-PMH at work in TeachEngineering 1.0, point your browser to the following URL (give it a few seconds to generate results; the program on the other side must do all the work):

https://cob-te-web.business.oregonstate.edu/cgi-bin/OAI-XMLFile-2.1/XMLFile/tecollection-set/oai.pl?verb=ListRecords&metadataPrefix=nsdl_dc

Take a look at the returned XML and notice the following:

- The OAI-PMH's XSD is located at http://www.openarchives.org/OAI/2.0/OAI-PMH.xsd

- Collapse each of the `<record>`s (click on the '-' sign in front of each of them). You will notice that only 20 records are served as part of this request. However, if you look at the `<resumptionToken>` tag at the bottom of the XML, you will see that the *completeListSize*=1581.
- The reason for serving only the first 20 records is the same as Google serving only the first 10 search results when doing a Google search, namely that serving all records at once can easily bog down communication channels. Hence, if, in OAI-PMH, you want additional results, you have to issue follow-up requests requesting the next set of 20 results:*https://cob-te-web.business.oregonstate.edu/cgi-bin/OAI-XMLFile-2.1/XMLFile/tecollection-set/oai.pl?verb=ListRecords&resumptionToken=nsdl!!!nsdl_dc!20*[5]
- Open up the first (top) `<record>` and take a look at its content. Notice how for this TeachEngineering items a variety of metadata are provided; *e.g.*, `<dc:title>`, `<dc:creator>`, `<dc:description>`, `<dc:publisher>`, *etc.*
- Notice the *dc* prefix in each of elements listed in the previous points. This stands for *Dublin Core*, a widely accepted and used standard for describing library holdings.

Serving Different XML Formats with XSLT

Let us take it one last step further. We just saw how, in the model for NSDL data hub harvesting, TE (1.0) provides information on its collection's holdings in XML over HTTP (using OAI-PMH), in Dublin Core (dc) format (indeed, quite a mouthful). But how about providing information about these same resources in other formats? For instance, IEEE developed the Learning Object Model (IEEE-LOM) format which is different from Dublin Core in that it was developed not to capture generic library resource information, but to capture learning and pedagogy-related information. If we would now want to service IEEE-LOM requests in addition to NSDL-DC requests, would we have to develop a whole new and additional XML web service? Fortunately, the answer is 'no.. This requires some explanation. Consider the three components at work here:

1. HTTP: the information transfer medium
2. XML (DC, IEEE-LOM or something else): the metadata format
3. OAI-PMH: the protocol for requesting and extracting the XML over HTTP

Seen from this perspective, the difference between DC, IEEE-LOM or, for that matter, any other XML representation of resource metadata is the only variable one of the three and hence, if we could easily translate between one type of XML and another, and if indeed the consumer can process OAI-PMH, we should be in business.

As it happens, the widely accepted XML translation technology called Extensible Stylesheet Language

5. Please ignore the mysterious sequences of bangs (!) in the *resumptionToken* parameter. They are part of the darker recesses of OAI-PMH.

Transformations (XSLT) does just that: convert between different types of XML. The way it works is —at least in principle— both elegant and easy. All you need to do is formulate a set of rules which express the translation from one type of XML, for instance DC, to another, for instance IEEE-LOM. Next, you need a program which can execute these translations, called an 'XSLT processor.' Once you have these two, all you need to do is run the processor and point it to an XML file containg the original XML and it will output the information in the other XML version.

Exercise 3.8

Once again, consider our 'family' XML document at: *https://classes.business.oregonstate.edu/reitsma/family.xml*

```
  <?xml version="1.0" encoding="UTF-8"?>
<family>
  <person gender="male">
    <firstname>Don</firstname>
    <lastname>Hurst</lastname>
  </person>
  <person gender="female">
    <firstname>Mary</firstname>
    <lastname>Hurst</lastname>
  </person>
</family>
```

Now let us assume that we want to translate this into a form of XML which only holds the `<firstname>`s and ignores the `<lastname>`s; *i.e.*, something like the following:

```
  <?xml version="1.0" encoding="UTF-8"?>
<family>
  <person gender="male">Don</person>
  <person gender="female">Mary</person>
</family>
```

In other words, we must 'translate' the first form of XML to the second form of XML.

Now, consider a file containing the following XSLT translation rules:

```
  <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/family">
```

```
    <family>
      <xsl:apply-templates select="person"/>
    </family>
  </xsl:template>

  <xsl:template match="person">
    <person gender="{@gender}">
      <xsl:value-of select="firstname" />
    </person>
  </xsl:template>
</xsl:stylesheet>
```

The file contains two rules: the first (`<xsl:template match="/family">`) specifies that the translation of a `<family>` consists of the translation of each `<person>` within the `<family>`. The second (`<xsl:template match="person">`) says that only a `<person>`'s *gender* and `<firstname>` must be copied. However, the `<firstname>` tag itself should not be copied.

Running this XSLT on the original XML file, we should indeed get:

```
  <?xml version="1.0" encoding="UTF-8"?>
<family>
  <person gender="male">Don</person>
  <person gender="female">Mary</person>
</family>
```

Let us now try this.

1. In a new tab, point your web browser to https://www.freeformatter.com/xsl-transformer.html
2. Enter our *family.xml* code into the *Option 1: Copy-paste your XML document here* textbox.
3. Enter our XSLT code into the *Option 1: Copy-paste your XSL document here* textbox.
4. Click the *Transform XML* button and note the results in the *Transformed Document* frame

This approach, of course, suggests lots of other possibilities. Using the same technique, we can, for instance, translate from the original XML into HTML. All we have to do is modify the XSLT. Let us see if we can make Don and Mary show up in an HTML table by modifying the XSL:

```
  <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
<body>
  <h2>Family Members</h2>
```

```
  <table border="1">
    <tr bgcolor="red">
      <th style="text-align:left">First name</th>
      <th style="text-align:left">Last name</th>
    </tr>
    <xsl:for-each select="family/person">
    <tr>
      <td><xsl:value-of select="firstname"/></td>
      <td><xsl:value-of select="lastname"/></td>
    </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Try it and notice how we have now transformed from XML into HTML using XSLT (Dump the HTML to a file and pick it up with your web browser). Not bad, hey?

XSLT in TE 1.0

Going back to the DC/IEEE-LOM situation in TE 1.0, we might consider using DC as the base XML and writing an XSLT to translate from DC to IEEE-LOM. The problem with that approach, however, is that the DC format accommodates only a small set of attributes and, hence, many things about TeachEngineering documents cannot be easily coded in DC to start with.

This, however, suggests a different, and we think better approach. Rather than using DC or for that matter IEEE-LOM as the base XML, why not use our own TE-internal XML as a base and then use XSLT to translate to whatever format anybody ever wants? That way, we have an internal XML format that can accommodate anything we might ever have in a TE document, yet we can use XSLT technology to serve DC, IEEE-LOM or anything else. This is very much the same approach that a program such as Microsoft Excel uses to present identical data in different formats. For instance, in Excel, a number can be shown with different amounts of decimals, as a percent, as a currency, *etc*. Internally to the program there is only one representation of that number, but to the user this representation can be changed through a transform.

For TE, if the requester of that information can process OAI-PMH, we would have a nice XML—XSLT—OAI-PMH pipeline for serving anything as an XML web service.

So this is what we did in TE 1.0. Much to our fortune, the OAI-PMH functionality was available as the open source *XMLFile* package written in Perl by Hussein Suleman (2002) while at Virginia Tech. Since Suleman

had already provisioned the server with means to accommodate multiple XSLTs (nice!), all we had to do was write the various XSLTs to translate from the base XML into the requested formats.

References

Allen, P. (2006) Service Orientation: Winning Strategies and Best Practices. Cambridge University Press. Cambridge, UK.

Baldwin, A.A., Brown, C.E., Trinkle, B.S. (2006) XBRL: An Impacts Framework and Research Challenge. Journal of Emerging Technologies in Accounting, Vol. 3, pp. 97-116.

Berners-Lee, T, Hendler, J., Lassila, O. (2001) The Semantic Web. Scientific American. 284. 34-43.

Bosak, J., Bray, T. (1999) XML and the Second-Generation Web. Scientific American. May. 34-38, 40-43.

MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., Metz, R. (2006) Reference Model for Service-Oriented Architectures 1.0. Public Review Draft 2. OASIS. http://www.oasis-open.org/committees/download.php/18486/pr-2changes.pdf. Accessed: 12/2016.

Pulier, E., Taylor, H. (2006) *Understanding Enterprise* SOA. Manning. Greenwich, CT.

Suleman, Hossein (2002) XMLFile. Available: http://www.husseinsspace.com/research/projects/xmlfile/. Accessed: 06/2022.

Vasudevan, V. (2001) A Web Services Primer. A review of the emerging XML-based web services platform, examining the core components of SOAP, WSDL and UDDI.

W3C (2002) Web Services Activity Statement. http://www.w3.org/2002/ws/Activity.html. Accessed: 04/07/2008

W3C (2004) Web Services Glossary. http://www.w3.org/TR/ws-gloss/#defs. Accessed: 04/07/2008

4. TE 2.0 – JSON

JavaScript Object Notation (JSON) – The New XML

The previous chapter discussed how XML was an invention which made possible a great variety of programmatic use of web content. The new kid on the block, however, is an alternative, lighter-weight data interchange format known as JSON (pronounced: Jay-son).

Although JSON's history goes back almost as far as XML's, its recent rise as an alternative for XML stems from several factors:

- It is lightweight in that it has less overhead than XML (just take this for granted right now; we will explain this later).
- It is often —although not necessarily— less verbose (less 'wordy') than XML and, therefore, faster to transfer across networks.
- It is tightly linked with JavaScript, which has seen very rapid growth as the programming language for web browser-based processing.
- A growing number of databases support the storage and retrieval of data as JSON.

Before we consider each of these, let us first look at JSON as a means of representing information.

JSON, as XML, is a way of hierarchically representing data in that it uses the same tree-like structure to represent information in nested form. Table 1 shows the identical information in both XML and JSON (example taken from Wikipedia's JSON page).

Table 1. Identical data represented both in XML (left) and JSON (right)

XML

```
<person>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <age>25</age>
    <address>
        <streetAddress>21 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021</postalCode>
    </address>
    <phoneNumbers>
        <phoneNumber>
            <type>home</type>
            <number>212 555-1234</number>
        </phoneNumber>
        <phoneNumber>
            <type>fax</type>
            <number>646 555-4567</number>
        </phoneNumber>
    </phoneNumbers>
    <gender>
        <type>male</type>
    </gender>
</person>
```

JSON

```
{"person":
  { "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address":
      { "streetAddress": "21 2nd Street
        "city": "New York",
        "state": "NY",
        "postalCode": "10021" },
    "phoneNumber": [
      { "type": "home",
        "number": "212 555-1234" },
      { "type": "fax",
        "number": "646 555-4567" } ],
    "gender": { "type": "male" } } }
```

Consider the JSON. It is a data structure which consists of a single complex element (*person*) containing six sub elements: *firstName*, *lastName*, *age*, *address*, *phoneNumber*, and *gender*. Of these, *address* and *gender* are once again complex. *phoneNumber* is a list containing two complex elements.

The JSON representation looks very much like a JavaScript data structure. In fact, it actually is just such a structure, which we can illustrate with the following exercise.

Exercise 4.1

Store the following content in a file *foo.html* and pick it up with your browser.

```
<html>
```

```
<script language="javascript">
var foo = {
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
alert(foo.gender.type);
</script>
</html>
```

What did we just do? We declared a JavaScript variable `foo` and assigned it the JSON structure. Next, we passed `foo`'s `gender.type` to the JavaScript `alert()` method, which pops it up in your browser. So apparently, the JSON structure in Table 1 is perfect JavaScript code all by itself. Hence, it fits seamlessly and without parsing or special processing in a JavaScript program.

If we compare this code with the exercise in the previous chapter where we used JavaScript to parse XML, the advantage of using JSON over XML when working in JavaScript becomes clear. Since JSON is just JavaScript —at least on the data side of things— when working in JavaScript, no parsing or special processing of JSON is needed. We can just grab it, store it in a variable and we are ready to go.

Of course, it does not really matter whether the JSON is embedded in the JavaScript as in our example above or if we retrieve it from an external source. In the next exercise we do the latter.

Exercise 4.2

Point your browser at https://classes.business.oregonstate.edu/reitsma/person.html and note how it results in *John Smith* being echoed in your browser. Now look at the *person.html* source code:

```
  <html>

<p id="person"></p>

<script>
var request = new XMLHttpRequest();
request.overrideMimeType("application/json");
request.onreadystatechange = extract;
request.open("GET",
             "https://classes.business.oregonstate.edu/reitsma/person.json",
             true);
request.send();

function extract()
{
  if (request.readyState == 4 && request.status == 200)
  {
    var person = JSON.parse(request.responseText);
    document.getElementById("person").innerHTML =
        person.firstName + " " + person.lastName;
  }
}
</script>

</html>
```

Notice how this is pretty much exactly what we did in the exercise in the previous chapter when we retrieved XML from a web source and parsed it. Here we retrieved JSON from a web source (https://classes.business.oregonstate.edu/reitsma/person.json) and echoed some of its content.

Also note the call to `JSON.parse()`. The method takes in a string returned from the external web source and tries parsing it into a JavaScript structure. If the string represents valid JSON —as in our case— that will work just fine. We then assign that structure to the variable `person`:

```
  var person = JSON.parse(request.responseText);
```

As we did in the XML variant of this, we then extract information from that `person` and substitute it for the content of the HTML tag with `id="person"`:

```
document.getElementById("person").innerHTML =
   person.firstName + " " + person.lastName;
```

JSON in Python

Whereas JSON is particularly efficient to use in a JavaScript context, it can, just as XML, be used in other contexts as well. In fact, JSON has become such a common format for exposing and exchanging data across the web that many programming languages other than JavaScript can be used to consume or generate JSON. Let us, once again, extract John Smith from the external JSON web source, but this time using Python (3.*)

Exercise 4.3
Run the following Python (3.*) code:

```
  import requests
import json

#Request the JSON over HTTP
try:
  response = requests.get( \
    "https://classes.business.oregonstate.edu/reitsma/person.json")
except Exception as err:
  print("Error downloading JSON...\n\n", err)
  exit(1)

#Load the retrieved JSON into a JSON structure
json_data = json.loads(response.text)

#Since JSON objects are dictionaries, we can index into them by name.
first_name = json_data["firstName"]
last_name = json_data["lastName"]
street_address = json_data["address"]["streetAddress"]

print(first_name, last_name, "\n", street_address)
```

DTDs or XSDs for JSON: *JSON Schema*

In the previous chapter we discussed how DTDs and XSDs are used to declare the syntax of XML documents. We also discussed document validation as one of the functions of these specifications. You may therefore wonder whether or not a similar standard exists for JSON documents. Indeed, there is such a standard, namely JSON *Schema*, sponsored by the Internet Engineering Task Force (ITEF).

JSON Schema is heavily based on the approach taken by XML Schema. Just as XSDs are written in XML, so is JSON Schema written in JSON and just as for XML, the JSON Schema is self describing.

To provide a flavor of JSON Schema, we use the example from json-schema.org (2016) of a simple product catalog. Here is the JSON for the catalog (only two products included[1]):

```
[
  {
    "id": 2,
    "name": "An ice sculpture",
    "price": 12.50,
    "tags": ["cold", "ice"],
    "dimensions": {
      "length": 7.0,
      "width": 12.0,
      "height": 9.5
    },
    "warehouseLocation": {
      "latitude": -78.75,
      "longitude": 20.4
    }
  },
  {
    "id": 3,
    "name": "A blue mouse",
    "price": 25.50,
    "dimensions": {
      "length": 3.1,
      "width": 1.0,
      "height": 1.0
    },
    "warehouseLocation": {
      "latitude": 54.4,
```

1. In this example we ignore details such as the units on dimensional numbers. For instance, are product dimensions in feet, inches, centimeters? And how about product weight? Similarly, any warehouse would likely have some identifier associated with it rather than just longitude and latitude. Still, as an example of JSON/Schema, this works fine.

```
      "longitude": -32.7
    }
  }
]
```

Pretty straightforward so far (note: the product catalog is a list ([...])). Now let us take a look at its JSON Schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product set",
  "type": "array",
  "items": {
    "title": "Product",
    "type": "object",
    "properties": {
      "id": {
        "description": "The unique identifier for a product",
        "type": "number"
      },
      "name": {
        "type": "string"
      },
      "price": {
        "type": "number",
        "minimum": 0,
        "exclusiveMinimum": true
      },
      "tags": {
        "type": "array",
        "items": {
          "type": "string"
        },
        "minItems": 1,
        "uniqueItems": true
      },
      "dimensions": {
        "type": "object",
        "properties": {
          "length": {"type": "number"},
          "width": {"type": "number"},
          "height": {"type": "number"}
        },
        "required": ["length", "width", "height"]
```

```
      },
      "warehouseLocation": {
      "description": "Coordinates of the warehouse with the product",
      "$ref": "http://json-schema.org/geo"
      }
    },
    "required": ["id", "name", "price"]
  }
}
```

Studying this data structure, one quickly notices its 'programming' or 'programmatic' orientation. For instance, it declares variables to be of traditional data types such as *string*, *array* or *number*. Also, the variable *required* is a string array containing the strings "*id*", "*name*" and "*price*". This programmatic orientation makes JSON structures a little easier to parse than XML strings and, as mentioned before, since JSON is essentially JavaScript, makes JSON integrate seamlessly into JavaScript programs.

Discussion: XML *vs.* JSON; XSD *vs.* JSON Schema; Is This Just the Next Cycle?

In the previous section we introduced JSON Schema as a way to specify the syntax of a JSON document, just as XSD is a way to specify the syntax of an XML document. So one might ask: if JSON is essentially like XML and if it too requires a validation meta layer (JSON Schema), then what is its real advantage over XML, if any? Let us reconsider the (alleged) advantages of JSON mentioned at the start of this chapter:

- JSON is light weight: it has less overhead than XML. It is less verbose than XML and, therefore, faster to transfer across networks.
- It is tightly linked with JavaScript which is the *de facto* programming language for web browser-based computing.
- A growing number of databases support the storage and retrieval of data as JSON.

The tight linkage with JavaScript and the availability of fast databases which store data as JSON are clear JSON advantages. Using 'raw' JSON directly in our (JavaScript) programs eliminates a parsing step. Similarly, because JSON is so closely related to object-oriented data representation, JSON structures are easily (de)serializable in object-oriented programming languages other than JavaScript. With (de)serialization we mean the conversion of a JSON string into an object in memory (deserialization) or *vice versa* (serialization).

The availability of JSON databases is another important factor in the attractiveness of JSON. If we can just 'throw' JSON structures into a database and then have the database software search those structures for certain data elements, that can make life nice and easy, especially if we are willing and able to relax on the 'normal form' and integrity constraints we are so accustomed to in the relational world. There are, of course, XML databases as well, yet somehow, JSON seems to be the 'new kid in town,' quickly either replacing XML or providing an additional format for data exchange.

What about the 'lightweight' argument, though? It is true that XML seems more verbose. After all, in XML we must embed data in tags whereas in JSON there is no such requirement. To gain a rough idea of the relative sizes of XML *vs*. JSON data sets, we compared the sizes of a small series of data sets randomly collected (scout's honor!) from www.data.gov, available in both XML and JSON (Table 2). Except for the smallest of data sets, the XML sets are, on average, almost twice the size of the corresponding JSON sets.

Table 2: Comparison of XML and JSON data sets found at www.data.gov.

www.data.gov data set[2]	XML (bytes)	JSON (bytes)	XML/ JSON
data.consumerfinance.gov/api/views.xml{json}	132003	143515	.920
data.cdc.gov/api/views/ebbj-sh54/ rows.xml?accessType=DOWNLOAD	36055	51102	.706
data.cdc.gov/api/views/w9j2-ggv5/ rows.xml?accessType=DOWNLOAD	566948	352678	1.608
data.cdc.gov/api/views/fwns-azgu/ rows.xml?accessType=DOWNLOAD	5178244	2353920	2.200
data.montgomerycountymd.gov/api/views/ 4mse-ku6q/rows.xml?accessType=DOWNLOAD	1513181656	677424751	2.234
data.illinois.gov/api/views/t224-vrp2/ rows.xml?accessType=DOWNLOAD	630175	362823	1.737
data.oregon.gov/api/views/kgdq-26yj/ rows.xml?accessType=DOWNLOAD	1268018	707931	1.791
data.oregon.gov/api/views/c5a8-vfhd/ rows.xml?accessType=DOWNLOAD	500160	324444	1.542
data.ny.gov/api/views/rsxa-xf6b/ rows.xml?accessType=DOWNLOAD	140290178	71363706	1.966
data.ny.gov/api/views/e8ky-4vqe/ rows.xml?accessType=DOWNLOAD	875873994	329161433	2.661

Another dimension of 'weight' is the overhead: the extra load or burden associated with working with XML *vs*. JSON datasets. To be clear, neither XML nor JSON mandates the use of a Schema and, hence, one should not hold the existence of extensive XML schemas and the relative absence of JSON schemas as a relative JSON advantage. However, since XML is the older of the two technologies, it has a deeper penetration in organizational, business and governmental computing and, hence, its ecosystem of protocols for standardization and validation is quite encompassing. Examples of these are XSD but also protocols such as SOAP, WSDL and the now defunct UDDI which were aimed to make XML into a general and overarching data representation and data exchange mechanism. Elegant and general as these might be, they often resulted in highly complex and arcane data structures and made computing harder by adding more 'regulation' in the form of additional hurdles to take. These protocols can be experienced as constraints or 'overkill' by those who wish to rapidly develop an application without being encumbered with those 'governance' protocols.

Although the absence of these protocols from much of the (current) JSON ecosystem should, perhaps,

2. All URLs refer to the XML version of the data sets (**.xml*). To retrieve the JSON versions, replace *.xml* with *.json*.

not be considered as a proper argument in the XML *vs.* JSON debate, the relative *laissez faire* climate of the JSON world does seem to promote developers to move away from XML and toward JSON. Will this trend continue? That remains to be seen. As JSON becomes more entrenched in organizational, business and governmental computing and data exchanges, the desire for validation, translation and specification of rich and complex data structures will likely increase. This could then well drive 'regulation' in the form of protocol specification, and this implies programming overhead in pretty much the same way it occurred for XML. Still, JSON's footprint in terms of byte size advantage, its database advantage and its (de)serialization advantage are real.

Perhaps the most likely outcome is that JSON and XML both remain relevant and complementary technologies, with JSON being most prevalent in gluing together modern applications and XML being used for marking-up content and in data exchange scenarios where rigorous validation (XSD) and transformation (XSLT) are called for.

Exercise 4.4: Who is putting out JSON services?

As with XML web services, JSON services do not typically come across people's web browser. The reason, of course, is that those services are meant for machines (programs) to be consumed, not people. Still, with a little googling it is pretty easy to find some of the many JSON services currently in function.

One example are many of the data sets offered by the USA government on sites such as data.gov, cdc.gov, or census.gov.

- Ask the API of the US census Bureau for information on an address in Corvallis, Oregon

  `https://geocoding.geo.census.gov/geocoder/locations/address?street=120+NW+4th+Street`

- Request USA life expectancy data from the CDC

  `https://data.cdc.gov/api/views/w9j2-ggv5/rows.json`

- TeachEngineering too exposes some of its data as JSON. For instance, to see a complete list of all of TeachEngineering's resources in JSON, point your browser to

  `https://www.teachengineering.org/api/standards/VisualizationExport`

Notice that, if your web browser is set up for this, it recognizes the returned results as JSON and renders them accordingly. If your browser puts out the returned JSON as one long, unformatted string, it has not been set up to recognize JSON. There are several ways to make this string more readable though. Some of these are web browser specific, such as the *JSONView* plugin for the Firefox browser or *JSON Viewer* for the Chrome browser. A browser-agnostic, on-line service is available at http://jsonviewer.stack.hu/. Simply enter any JSON string in the site's *Text* tab and click on the *Viewer* tab.

The data sets listed in Table 2 show that static JSON sets are also increasingly available. One only needs to peruse the thousands of data sets available through www.data.gov to notice that increasingly, JSON is one of the formats offered for data retrieval.

TeachEngineering (TE 2.0) Resources as JSON structures

In the previous chapter on XML we saw that TE 1.0 resources were stored as XML structures. This worked fine and served some valuable purposes such as resource rendering, document validation and metadata provisioning. In case you must refresh your memory on TE 1.0 XML, refer back to the TE 1.0 example.

In TE 2.0, however, we switched from XML to JSON. The switch was motivated by the reasons mentioned in the opening paragraph of this chapter: JSON is lightweight, quick to transport and perhaps most important, the recent availability of JSON-based databases which allow for fast storage and retrieval of JSON-based data.

Take a look at the (partial and abbreviated!!) JSON representation of the same 'intraocular' resource we looked at in XML:

```
{
  "Header":"<p><img data-url=
  \"mis_/activities/mis_eyes/mis_eyes_lesson01_activity1_image1web.jpg\"
  data-rights=\"Apple Valley Eye Care. Used with permission"
  data-caption=\"As seen in the image, irreparable vision loss can occur
  in persons with glaucoma.\"
  alt=\"A photograph of two young girls looking at a camera.
  The edges of the image have a black vignette—a loss in clarity towards
  the corners and sides of an image—which portrays what is seen when
  damage to the optic nerve has occurred due to the effects of glaucoma.\"
  /></p>",
  "Dependencies":[
    {
      "Url":"mis_eyes_lesson01",
      "Description":null,
      "Text":"These Eyes!",
      "LinkType":"Lesson"
    }
  ],
  "Time":{
    "TotalMinutes":350,
    "Details":"<p>(seven 50-minute class periods)</p>"
  },
  "GroupSize":3,
  "Cost":{
    "Amount":0.3,
    "Details":"<p>Students use online web quest (free), 3D modeling app (free)
  and a 3D printer (or modeling clay) to design and create prototypes.</p>"
  },
  "EngineeringConnection":"<p>Biomedical engineers rely on modeling to design
  and create prototypes for devices that may not yet be approved for testing.
```

In order to prepare for the cost of manufacturing a device, careful consideration goes into the potential constraints of that device. Using various software programs, engineers design and visualize the device they wish to create in order to determine whether the future device is worth the effort, time and expense.
Mirroring real-world engineers, in this activity, students play the role of engineers challenged to create intraocular pressure sensor prototypes to measure pressure within the eyes of people with glaucoma.</p>",
"EngineeringCategoryType":"Category2EngineeringAnalysisOrPartialDesign",
"Keywords":[
  "3D printer",
  "3D printing",
  "at-scale modeling",
  "biomedical"
],
"EducationalStandards":[
  {
    "Id":"http://asn.jesandco.org/resources/S113010D",
    "StandardsDocumentId":"http://asn.jesandco.org/resources/D1000332",
    "Jurisdiction":"Michigan",
    "Subject":"Science",
    "ListId":null,
    "Description":[
      "Science Processes",
      "Reflection and Social Implications",
      "K-7 Standard S.RS: Develop an understanding that claims and evidence for their scientific merit should be analyzed. Understand how scientists decide what constitutes scientific knowledge. Develop an understanding of the importance of reflection on scientific knowledge and its application to new situations to better understand the role of science in society and technology.",
      "Reflecting on knowledge is the application of scientific knowledge to new and different situations. Reflecting on knowledge requires careful analysis of evidence that guides decision-making and the application of science throughout history and within society.",
      "Design solutions to problems using technology."
    ],
    "GradeLowerBound":7,
    "GradeUpperBound":7,
    "StatementNotation":"S.RS.07.16",
    "AlternateStatementNotation":"S.RS.07.16"
  },
  {
    "Id":"http://asn.jesandco.org/resources/S114173E",
    "StandardsDocumentId":"http://asn.jesandco.org/resources/D10003E9",

```
    "Jurisdiction":"International Technology and Engineering Educators Association",
    "Subject":"Technology",
    "ListId":"E.",
    "Description":[
      "Design",
      "Students will develop an understanding of the attributes of design.",
      "In order to realize the attributes of design, students should learn that:",
      "Design is a creative planning process that leads to useful products and systems."
    ],
    "GradeLowerBound":6,
    "GradeUpperBound":8,
    "StatementNotation":null,
    "AlternateStatementNotation":null
  },
```


Comparing things with the TE 1.0 XML, things look quite similar. However, there are a few important differences:

1. Whereas in the XML version of the resources only a reference to an educational standard was kept, such as *S113010D* or *S114173E*, in the JSON version not only the identifiers, but all the properties of the standard —description, grade levels, *etc.*— are stored as well. To anyone trained in and used to relational database modeling and so-called 'normal form' this raises a big red flag as it implies a potential for a lot(!) of data duplication because each time that the standard appears in a resource its entire content is stored in that resource. How likely is this to happen? Table 3 shows a tally of only the ten most-referenced standards and the number of times they occur. Just for these ten standards this results in 1,121 duplications. Add to that, that in TeachEngineering more than 1,200 different standards are used more than once and we can see why relationally trained system designers frown when they notice this. Interesting observation: proponents of non-relational (NoSQL) databases refer to this practice of duplicating data as 'hydrating.' Those proponents would label the above case —each standard contains its entire data, regardless of how many other standards share that same data— as being 'fully hydrated.'

Table 3: Ten most referenced K-12 education standards in TeachEngineering and their number of occurrences.

Standard	Number of occurrences in TeachEngineering
S11416DD	176
S11434D3	140
S2454468	127
S2454533	125
S2454534	117
S11416DA	107
S2454469	92
S11416D0	83
S1143549	81
S114174D	73
Total number of duplicates in the ten most referenced standards	1,121

It is important, however, to realize that this difference between the TE 1.0 XML representation and the TE 2.0 JSON representation is not at all related to differences in how XML and JSON represent information. After all, the designers of TE 2.0 could have easily chosen to include only the standard references in the JSON representation and leave out the standards' contents. Choosing to include the standards' contents in the resources and hence having to accept its consequences in the form of quite extensive data duplication, therefore, was entirely an architectural decision. We discuss this decision in the next chapter on document databases.

2. A second difference between the XML and JSON representation is that certain members of the JSON representation seem to contain explicit HTML. For instance, the text of the *Header* section of the activity JSON above contains HTML's `<p>` and `<img>` tags. On first inspection, this may seem strange as in the previous chapter we celebrated the value of text-based web services such as the ones based on XML (and hence, JSON), because they liberated developers from the use of HTML, a language meant for formatting rather than content description. Why then, one may ask, introduce formatting instructions in the content description? Interestingly, when we take a look at the TE 1.0 XML content of that same header, we see something similar:

```
<header>
```

```
  <text_section>
    <text_block format="text">
      <text_element>
        <image description="A photograph of two young girls
            looking at a camera. The edges of the image have
            a black vignette—a loss in clarity towards the
            corners and sides of an image—which portrays what
            is seen when damage to the optic nerve has
            occurred due to the effects of glaucoma."
          url="mis_eyes_lesson01_activity1_image1web.jpg"
          rights="Apple Valley Eye Care. Used with permission.
http://aveyecare.com/photolibrary_rf_photo_of_glaucoma_vision.jpg"
          caption="As seen in the image, irreparable vision
          loss can occur in persons with glaucoma."
          />
      </text_element>
    </text_block>
  </text_section>
</header>
```

Clearly, in both cases we see formatting instructions included in the content descriptions. In the JSON case, the instructions are pure HTML whereas in the XML case they are XML-based elements conveying the same information. So what is going on here? Why this re-mixing of content and formatting of the XML and JSON after all this work in the late 1990s and early 2000s to separate them? The reason is subtle but not uncommon. When looking at TeachEngineering pages, we see that all resources of the same kind —lessons, activities, sprinkles— all have the same basic layout. Yet not all resources from the same type are precisely the same. For instance, some resources have more images than others and some center certain sections of text whereas others do not. Since the curriculum authors have some freedom to layout contents within the structural constraints of the collection, the formatting stored in both the XML and JSON representations is that specified by the resource authors and must be considered intrinsic to the resource's content.

Up to this point we have seen some of the differences and similarities between XML and JSON. On the face of it, the differences may seem hardly significant enough to warrant a wholesale switch from XML to JSON. Sure, JSON is perhaps a little faster and is perhaps easier to work with in JavaScript. The perspective changes quite dramatically, however, when we consider the integration of JSON and the new generation of NoSQL document databases, especially the JSON-based ones. That is the topic of our next chapter.

References

JSON-schema.org (2016) *Example data*. http://json-schema.org/example1.html. Accessed: 12/2016 (no longer available)