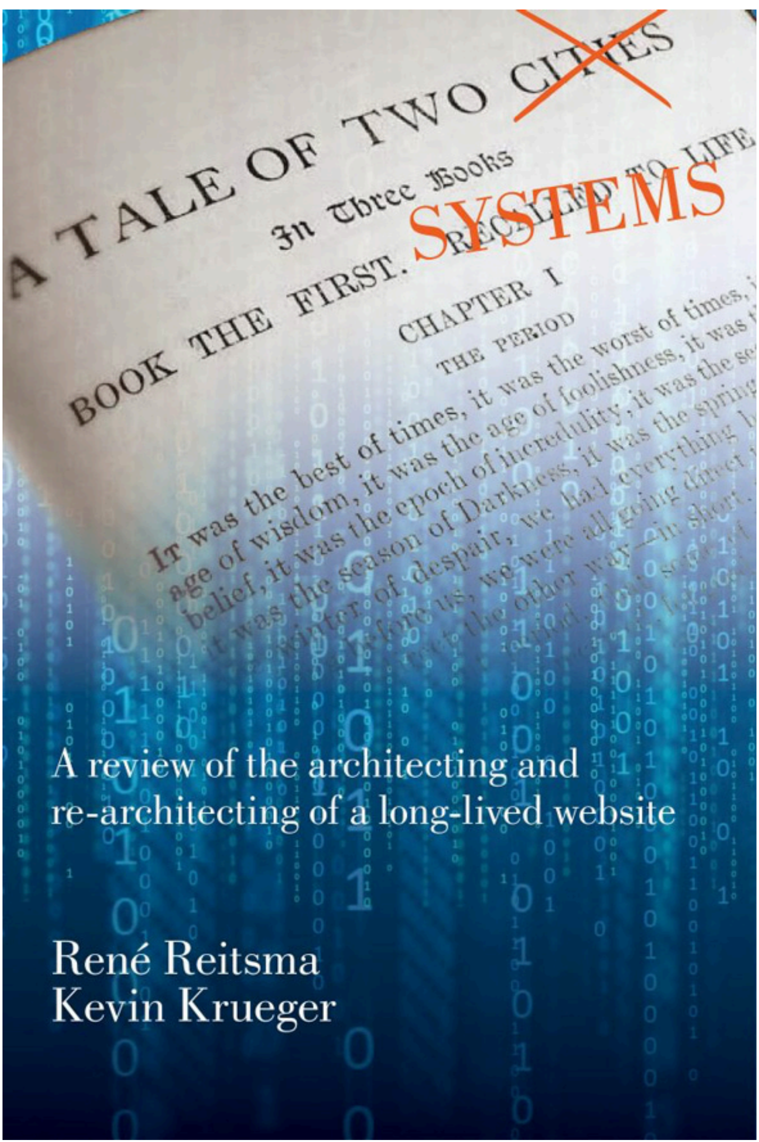
A Tale of Two Systems


A TALE OF TWO CITIES
SYSTEMS
In Three Books
BOOK THE FIRST.
CHAPTER I
THE PERIOD
A review of the architecting and
re-architecting of a long-lived website
René Reitsma
Kevin Krueger

A Tale of Two Systems by René Reitsma and Kevin Krueger is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License, except where otherwise noted.

Download for free at https://open.oregonstate.education/taleoftwosystems

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

Suggest a correction

Contents

	Preface	vii
	About the Authors	xii
	Acknowledgements	xv
1.	TeachEngineering (TE) Overview	1
2.	Why Build (Twice!) Instead of Buy, Rent or Open Source?	14
3.	TE 1.0 – XML	24
4.	TE 2.0 – JSON	65
5.	Relational (TE 1.0) vs. NoSQL (TE 2.0)	92
6.	Document Accessioning	139
7.	Why Build Revisited	157
8.	The Develop... Test... Build... Deploy Cycle	162
	Appendix A: When Editing Code Files, Use a Text Editor; Not(!) a Word Processor	175
	Appendix B: (Unintended?) Denial of Service Attack	180
	Appendix C: Fake Link Requests	186
	Appendix D: I am robot...	194
	Creative Commons License	212
	Recommended Citations	213
	Versioning	216

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 the times when people wrote their programs in COBOL on green screens.

Older, so called 'legacy' systems or applications 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 enjoyed lasting support from its growing user community and its sponsors. These 13 years, however, cover 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, certainly since the last five years or so, a rapidly growing emphasis on usability and graphic design, partly because of the need to move applications into the mobile domain, 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 *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, codechef.com 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

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 3.5 million users, has recently been rebuilt on a new architecture and remains as one of fewer than 100 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 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, 16 years later, TeachEngineering is still supported by NSF and a variety of other funders. It has grown to more than 1,500 curricular items contributed by many universities and programs and enjoys a patronage of over 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 of lots of 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 Document Collection

A quick look at TeachEngineering curriculum (https://www.teachengineering.org/curriculum/browse) reveals how the library is structured. It consists of a collection of documents, each of which is of one of four types: *activity*, *lesson*, *curricular unit* and *sprinkle*. *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 on a so-called *curricular unit*. *Sprinkles* 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 sprinkles are in both English and Spanish.

All documents 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 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).

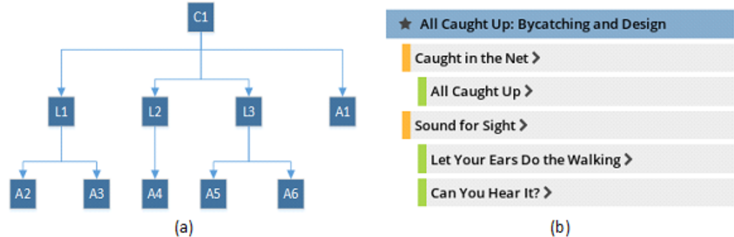


Figure 2: *Hierarchical structure of TE documents. (a) General example. The curricular unit C1 has three lessons (L1-L3) and six activities (A1-A6). Five activities (A2-A6) reside on lessons and one (A1) resides directly on the unit. (b) The All Caught Up curricular unit consists of two lessons and three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

As of January 2020, TeachEngineering has 89 curricular units, 516 lessons and 1059 activities.

Controlled Document 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 had to be imposed on each and every document. 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 document 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 byte-size integers.

Putting such strict structural and data type constraints on document content accomplishes two things. First, it allows the construction of a collection of documents 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 document ingestion and registration –a process known as *document accessioning*– quality control, document indexing, and metadata generation.

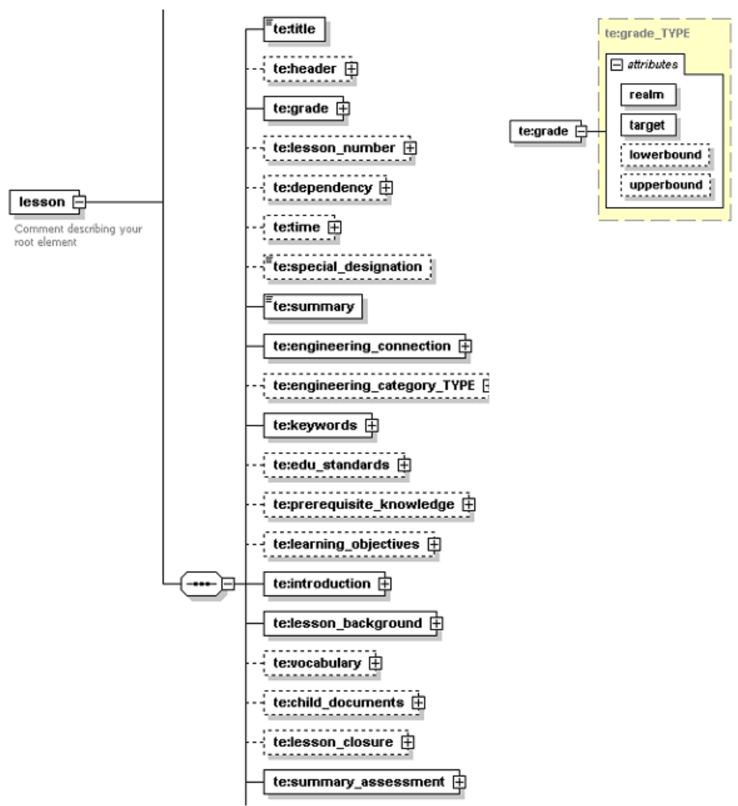


Figure 4: *Partial structure and content of a lesson. Solid-line rectangles indicate mandatory components; dashed lines indicate optional components (graphic generated with XMLSpy).[footnote]The document structure as displayed here is that of TE 1.0. In TE 2.0 some of these components were dropped and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

The latter point —automatic generation of metadata— is important as it is a classic bottleneck in the population of digital document repositories. With metadata, we mean data about a document rather than the document's content. 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 classic problem that such data entry is boring, time consuming, is error prone and must be reviewed and possibly redone when a document changes. 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 a lot of 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.

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 'standards.' In the USA the determination and authoring of these standards is the authority of individual states and within the states sometimes of individual districts. Although attempts at harmonizing standards across states —we refer to the Common Core for Mathematics and English and the Next Generation Science Standards initiatives (Conley, 2014; NRC, 2011)— have met with some success, 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 or write their own variations of those standards. 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 approximately 65,000 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 document which standards are supported by (align with) which document was not one of our core competencies, we hoped to acquire this service elsewhere.

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 documents 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 each TeachEngineering document 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 Document Accessioning

Although invisible to TeachEngineering users, TeachEngineering documents 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 document 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 document must be formatted to fit the TeachEngineering document template. This process changed quite dramatically between TE 1.0 and 2.0. Finally, the now formatted document must be registered to the collection so that it can be searched, displayed, saved as a bookmark, commented on, rated, *etc*. Chapter 6 covers document 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

SQL 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 5-10 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 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 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 documents 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 about 3 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, Rent or Open Source?

Build or Buy?

Not very long ago, in-house 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 such a system or many of its 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 15 or 20 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

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

computing and programming, an ever faster growing supply of both complete application systems and system components, both complex and powerful as well as elementary, have 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 it. 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 third parties.

It is therefore true that, more than in the past, we should look around for existing products 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, takes 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

(very minimal) version loosely based on Sun's Pixrect library.

or temporary advantage with that product will be harder because of the later adoption. Since, when looking for building blocks we tend to 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 rely on. 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.

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 documents 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 (and still uses) 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*

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 over– the decision to build rather than to buy.

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.

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 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

3. To detect if a website is Drupal based, point your Web browser to the website and display the page's HTML

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 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 in that each document follows one of four 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).

source code (view source code). Drupal pages contain the *generator* meta tag with its *content* attribute set to *Drupal x* where *x* is the Drupal version number.

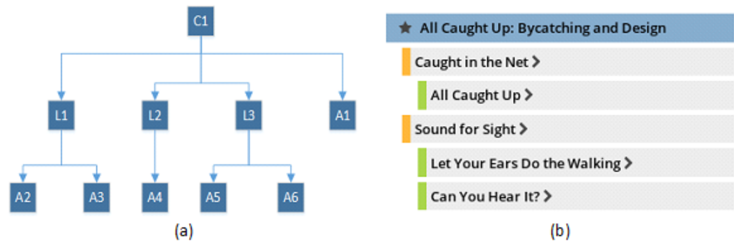


Figure 1: *Hierarchical structure of TE documents. (a) General example. The curricular unit C1 has three lessons (L1-L3) and six activities (A1-A6). Five activities (A2-A6) reside on lessons and one (A1) resides directly on the unit. (b) The All Caught Up curricular unit consists of two lessons and three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

It would of course be 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* lists 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

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

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 with professional software developers, the reverse might be the case.

Finally, the three most popular content management systems are 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 has 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 stack to first learn and master, then support and maintain. While there are 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.

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 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 it 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.

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 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 JSON in TE 2.0.

Coding Content with XML

One of the more influential advances in modern-day electronic data exchange, and one which caused web-based data exchanges, aka web services 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 a 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 —machines other than our own— access to those materials so that these machines can extract information from them. In other words, we want to offer a web service. 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
- Copyright(s)

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
  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
  eyes helps us see a larger area than just one eye and with tw
  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>` (line 24) or `<title>` (line 2). Hence, to find which educational standards this lesson supports, all we have to do is find the `<edu_standards>` tag (line 16-19) 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. It sees, for instance, that several `<keyword>`s are contained within the `<keywords>` tag and that there is regular text within each `<keyword>`. The same applies to the `<edu_standards>` and `<edu_standard>` tags.

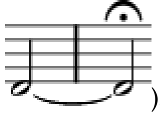
The above XML format is rather rigid and not entirely pleasant for us humans to read. However, it is this rigid notion of information items placed inside these tags and tags themselves placed within other tags that provides three essential advantages for machine-based reading:

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 of us 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 flatted ( ).
- 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) 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 ().

Now note that for humans 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 formatted 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>
    </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"/>
```

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.

```
      <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]

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.

$$s_N = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

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>
```

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.

```
      <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 web services, programs which serve

XML or JSON content over the web, is that they are self-describing. For XML, this takes the form of 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3c.org/2001/XMLSchema">
<xs:element name="lesson">
  <xs:complexType>
    <xs:sequence>

      <!--semantics for the <grade> element-->
      <xs:element name="grade">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:unsignedByte">
```

```
        <xs:attribute name="lowerBound" type="xs:unsigned
          use="optional"/>
        <xs:attribute name="upperBound" type="xs:unsigned
          use="optional"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<!--semantics for the <time> element-->
<xs:element name="time">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:float">
        <xs:attribute name="unit" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="minutes"/>
              <xs:enumeration value="hours"/>
              <xs:enumeration value="days"/>
              <xs:enumeration value="weeks"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<!--semantics for the <keywords> element-->
<xs:element name="keywords">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="keyword" type="string"
      maxOccurs="unbounded"/>
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
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 28-32).

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 XML*Schema*.

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 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</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 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 lesson document 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. However, if, as a data provider you make XML available to others, it is good practice to formulate 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 http://faculty.bus.oregonstate.edu/reitsma/family.xml. Notice that we have a small XML dataset 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 assume that we want to write a program which can automatically pick up this data set and extract the family members 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 each of the following three examples —PHP, C# and JavaScript— 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.

Exercise 3.4

Requesting XML over the web (HTTP) and extracting data from it using PHP

Store the following in a file called *family.php*:

```
<?php

//request the xml from the remote server and load it into a
$my_xml=simplexml_load_file(
"http://faculty.bus.oregonstate.edu/reitsma/family.xml") or
die("Error: Cannot create object");
```

```
//extract the persons and print their firstname and lastname
foreach($my_xml->children() as $persons)
{
  echo $persons->firstname . " ";
  echo $persons->lastname . "\n";
}

?>
```

Run this program:

- If you have PHP installed on your machine: on the command line:

  ```
  >php family.php
  ```

- at phpfiddle.org: paste the code and run.

Output:

- On the command line:

  ```
  Don Hurst
  Mary Hurst
  ```

- On phpfiddle.org:

  ```
  Don Hurst Mary Hurst
  ```

Question: Why would the command-line output be different from the *phpfiddle.org* output? Hint: note how the PHP program inserts a "*n*" between the two names. However, a newline in HTML is represented with a "*
" or "<p>*" tag.

Exercise 3.5

Requesting XML over the web (HTTP) and extracting data from it using C#

Consider the following C# program:

```
  using System;
using System.IO;
using System.Xml;

class Foo
{
  static void Main()
  {
    XmlDocument xmlD;
    XmlNodeList person_nodeList;

    //Create the XML Document
    xmlD = new XmlDocument();

    xmlD.Load(
    "http://faculty.bus.oregonstate.edu/reitsma/family.xml"
    );

    //Get the list of name nodes
    person_nodeList = xmlD.SelectNodes("/family/person");

    //Loop through the nodes
    foreach (XmlNode name_node in person_nodeList)
    {
      //Get the 'firstName' element value
      string firstNameValue =
```

```
        name_node.ChildNodes.Item(0).InnerText;

        //Get the 'lastName' element value
        string lastNameValue =
        name_node.ChildNodes.Item(1).InnerText;

        //Write result to the console
        Console.Write(firstNameValue + " " + lastNameValue);
        Console.WriteLine();
      }

    } // end Main()
  } // end class Foo
```

If you have Visual Studio on your machine:

1. Store this code in a file with a *.cs* extension; *e.g.*, *foo.cs*.
2. *Start Visual Studio* → *Visual Studio Tools Developer Command Prompt for VS*[3]
3. On the resultant command prompt, run the following commands:
   1. `>cd file_path_to_the_foo.cs_file`
   2. `>csc foo.cs` (compile the program)

3. Microsoft frequently changes the name of this menu option and its location on the *Start* menu. Important, however, is to know that a normal command-line window (cmd) window will not work unless a path to the C# compiler (*csc.exe*) has been set.

3. `>foo` (run it)

In *dotnetfiddle.net*, set the *Project Type* to *Console*, enter the code in the source code window, and hit the *Run* button. Note that you may be prompted that `Main()` *must be declared public in a public class*. If so, change these lines as follows:

```
class Foo → public class Foo

static void Main() → public static void
Main()
```

Exercise 3.6
Requesting XML over the web (HTTP) and extracting data from it using JavaScript.
One of the advantages of JavaScript is that pretty much all web browsers have a JavaScript interpreter built in. Hence, there is nothing to install 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",
"http://faculty.bus.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.[4] To make this work, however, we installed the exact same code at *http://faculty.bus.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:http://faculty.bus.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

4. 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).

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 3.7

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

https://COB-TE-Web.bus.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. This makes much sense, as the page is meant for 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* has 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.bus.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, translates them into HTML and serves the HTML to the requester.

View_activity.php does some other things as well. 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.

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 65,000 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.[5]

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="http://purl.org/ASN/schema/core/"
xmlns:cc="http://creativecommons.org/ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:dcterms="http://purl.org/dc/terms/"
xmlns:foaf="http://xmlns.com/foaf/0.1/"
xmlns:gemq="http://purl.org/gem/qualifiers/"
xmlns:loc="http://www.loc.gov/loc.terms/relators/"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:skos="http://www.w3.org/2004/02/skos/core#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<cc:attributionURL
rdf:resource="http://asn.desire2learn.com/resources/D2627218" /
<dc:title xml:lang="en-US">South Dakota Science
Standards</dc:title>
```

5. 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.

```
<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="http://purl.org/dc/terms/W3CDTF">
2015-05-19</asn:repositoryDate>

<asn:Statement
rdf:about="http://asn.desire2learn.com/resources/S2627378">
  <asn:statementNotation>K-PS2-1</asn:statementNotation>
  <dcterms:educationLevel
  rdf:resource="http://purl.org/ASN/scheme/ASNEducationLevel/K"
  <dcterms:subject
  rdf:resource="http://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="http://asn.desire2learn.com/resources/S2627379">
  <asn:statementNotation>K-PS2-1</asn:statementNotation>
  <dcterms:educationLevel
  rdf:resource="http://purl.org/ASN/scheme/ASNEducationLevel/K"
```

```
  <dcterms:subject
  rdf:resource="http://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:

```
http://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 3.8

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

```
http://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 its member libraries. For users —or *patrons* in library jargon— this would mean that they could come to NSDL and conduct faceted searches over all its member libraries without having to separately search all these libraries. With *faceted 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 10th grade teacher, if there is a 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 NSDL. Each of these members conducts its own search and reports back to the central agency 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 members.

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

As 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 quickly enough 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: what information should the member libraries submit, what form should the information be in, and 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? **Answer**: NSDL and its members should decide on a standard set of data items representing member holdings.
- **Question**: What form should the information be in? **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? **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 3.9

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 a lt of work):

https://COB-TE-Web.bus.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*=1551.
- 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.bus.oregonstate.edu/cgi-bin/OAI-XMLFile-2.1/XMLFile/tecollection-set/oai.pl?verb=ListRecords&resumptionToken=nsdl!!!nsdl_dc!20*[6]

- 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 or collection 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 provides information on its collection's holdings in XML over HTTP (using OAI-PMH), in Dublin Core (dc) format (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 item 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 web service? Fortunately, the answer is 'no,' if the IEEE-LOM consumer can

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

harvest OAI-PMH. 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 meta data carrier
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 item metadata is the only variable 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 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 with the original XML and, if all is well, it will output the information in the other XML version.

Exercise 3.10

Once again, consider our 'family' XML document at:
http://faculty.bus.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 the following file with 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 the `<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. Point your web browser to http://www.utilities-online.info/xsltransformation.
2. Enter our *family.xml* code into the XML frame.

3. Remove the line <?*xml version=”1.0″ encoding=”UTF-8″?*>
4. Enter our XSLT code into the XSL frame.
5. Remove the line <?*xml version=”1.0″ encoding=”UTF-8″?*>
6. Test both codes for validity (check buttons below the frames). If you get an error on the XSL syntax, check the double quotes (sometimes cutting and pasting quotes and double quotes across platforms causes problems).
7. Click the *Transform XML with* XSL button and note the results in the *Results* 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 Excel uses to present identical data in different formats; for instance, a number can be shown with different amounts of decimals, as a percent, *etc*. Internally to the program there is only one representation, but to the user this representation can be changed though a transform.

If the requester of that information can process OAI-PMH, we 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.dlib.vt.edu/projects/OAI/software/oai-file/oai-file.html. Accessed: 12/2016 (no longer available).

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

XML was an invention which made possible a great variety of programmatic use of web content. The new kid on the block, however, is a 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 light weight 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. The same data represented both in XML (left) and JSON (rig

h
t
)

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

```
{
    "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 ([json]{}[/json]) containing six sub elements: [json]firstName[/json] [json]lastName[/json], [json]age[/json], [json]address{}[/json], [json]phoneNumber[][/json] and [json]gender{}[/json]. Of these, [json]address{}[/json] and [json]gender{}[/json] are once again complex. [json]phoneNumber[][/json] is an array ([json][][/json]) 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 prove with the following exercise.

Exercise 4.1

Store the following content into 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. Let us do the latter.

Exercise 4.2

Point your browser at http://faculty.bus.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",
    "http://faculty.bus.oregonstate.edu/reitsma/person.js",
    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 (http://faculty.bus.oregonstate.edu/reitsma/person.js) 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 PHP and C#

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 PHP and C#.

Exercise 4.3

At *phpfiddle.org* enter the following code in the *CodeSpace* and run it. Nice!

```
<?php

$json = file_get_contents(
    "http://faculty.bus.oregonstate.edu/reitsma/person.js"
);
$person = json_decode($json, true);
echo $person['firstName'] . " " . $person['lastName'];

?>
```

Exercise 4.4

To do this in C#, we must have access to some JSON processing software first. Since at the time of this writing, the .NET *System.Json* Namespace had become defunct, for this example we are relying on *Newtonsoft.Json* (aka *Json.NET*). However, in order to prevent you from having to install this component on your machine, we will run this example in *dotnetfiddle.net*.

1. Enter the following C# code at *dotnetfiddle.net*.

   ```
   using System;
   ```

```
using System.Net;
using Newtonsoft.Json.Linq;

public class Foo
{
  public static void Main()
  {
    string json = new
WebClient().DownloadString(
"http://faculty.bus.oregonstate.edu/reitsma/person.js"
);

    JObject person = JObject.Parse(json);

    string firstNameValue =
        person.GetValue("firstName").Value<string>();
    string lastNameValue =
        person.GetValue("lastName").Value<string>();

    //Write result to the console
    Console.Write(firstNameValue + " " + lastNameValue);
    Console.WriteLine();

  } // end Main()
} // end class Foo
```

2. In the textbox for *NuGet Packages* do a search for (type) *json* and select the latest stable version of *Newtonsoft.Json.*
3. Run

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,
```

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.

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

Pretty straightforward so far (note: the product catalog is an array ([…])). 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 pro
      "$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), 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 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.

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 'light weight' 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. One aspect of this concerns the programming effort required to process data sets. Whereas a very rich array of tools and protocols is available to create, manipulate and parse XML data sets, fewer such tools are available for JSON. The JSON toolset, however, is rapidly growing and may already have caught up. Another aspect concerns required overhead in terms of XSDs or Schemas. To be clear, neither

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

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 much 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, not be considered as a proper argument in the XML *vs.* JSON debate, the relative *laissez faire* climate of the current 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 4.5: Who is putting out JSON services?

As with XML web services, JSON services do not typically come across the Web browser of people. 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 is Yahoo Finance's API for requesting stock quotes. The API can generate both XML and JSON; all you have to do is tell it which one you want.

- Ask the API for a stock quote of YAHOO (ticker symbol YHOO) as XML:

```
https://query.yahooapis.com/v1/public/yql?q=select * from yahoo.finance.quotes where symbol in ('YHOO')&format=xml&env=store://datatables.org/alltableswithkeys
```

- Next, request the same information as JSON

```
https://query.yahooapis.com/v1/public/yql?q=select * from yahoo.finance.quotes where symbol in ('YHOO')&format=json&env=store://datatables.org/alltableswithkeys
```

Although not of central importance here, note how the URL contains a parameter *q* which is set to a query in Yahoo Query Language (YQL):

```
select * from yahoo.finance.quotes where symbol in ('YHOO'
```

Also note that whereas your web browser recognizes the return from the XML service as XML and automatically displays the returned code in a form friendly for human reading, it might not render the JSON at all. Instead, it just shows it as an unstructured string. There are several ways to make this string more readable though. Some of these are web browser specific, such as the JSON*View* 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. It is interesting, perhaps, to see how this penetration/adoption differs in different domains, at least for now. For instance, many governmental data sets related to 'Health' are available in both JSON and XML format, yet of the 278 'Energy' data sets (01/06/2017), only seven were available as XML and zero as JSON. A similar situation applies to the 'Agriculture' data sets.

TeachEngineering (TE 2.0) Documents as JSON structures

In the previous chapter on XML we saw that TE 1.0 documents were stored as XML structures. This worked fine and served some

valuable purposes such as document 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 light weight, 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' document we looked at in XML:

```
{
  "Header":"<p><img data-url=
  \"mis_/activities/mis_eyes/mis_eyes_lesson01_activity1_image1
  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 documents 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 document, its entire content is stored in that document. 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 raise their eyebrows when they notice this.

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 documents 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 wha
            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 permissio
http://aveyecare.com/photolibrary_rf_photo_of_glaucoma_visic
          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 documents of the same kind —lessons, activities, sprinkles— all have the same basic layout. Yet not all documents from the same type are precisely the same. For instance, some documents have more images than others and some documents 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 document authors and must be considered intrinsic to the document'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)

5. Relational (TE 1.0) vs. NoSQL (TE 2.0)

Introduction: Relational is no Longer the Default MO

Whereas the coding (programming) side of system development has witnessed more or less constant innovation over the years, the data management side of things has long been dominated by the relational model, originally developed by E.F. Codd in the 1970s while at IBM (Codd 1970, 1982). One (imperfect) indicator of the dominance and penetration of the relational model in business IT systems is the share of coverage the relational *vs.* NoSQL database technologies receive in (Business) Information System Analysis and Design textbooks. Table 1 contains an inventory of a few of those books, namely the ones sitting on the shelves of one of us. Not a scientific sample, but telling nonetheless.

Table 1: Coverage of relational vs. NoSQL in some System Analysis

Textbook
Valacich, J.S., George, J.F., Hofer, J.A. (2015) *Essentials of Systems Analysis and Design*. Pear
Satzinger, J., Jackson, R., Burd, S. (2016) *Systems Analysis and Design in a Changing World*. Learning.
Tilley, S., Rosenblatt, H. (2017) *Systems Analysis and Design*. CENGAGE Learning.
Dennis, A., Wixom, B.H., Roth, R.A. (2014) *System Analysis and Des*ign. 6th ed. John Wiley &
Dennis, A., Wixom, B.H., Tegarden, D. (2012) *System Analysis & Design with UML Version 2.* Sons.
Dennis, A., Wixom, B.H., Tegarden, D. (2015) *System Analysis & Design. An Object-Oriented UML. 5th ed.* John Wiley & Sons.
Coronel C., Morris, S. (2016) *Database Systems. Design, Implementation and Management*. C learning.

To be clear, pointing out the almost total absence of NoSQL coverage in these texts is not meant as a critique of these texts. The relational database remains a dominant work horse of transaction processing and hence, remains at the heart of business computing. But whereas for a very long time it was essentially the only commonly available viable option for all of business computing—transaction processing or not—new, equally viable and commonly available non-relational alternatives for non-transaction processing are making quick inroads.

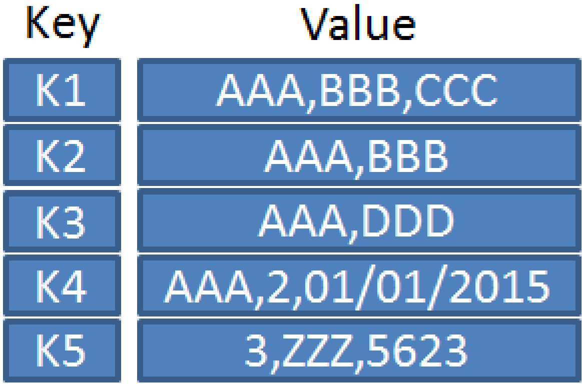


Figure 1: *Lookup structure of a key-value store (source: https://en.wikipedia.org/wiki/File:KeyValue.PNG)*

Of the six texts listed in Table 1, only Dennis, Wixom & Tegarden (2012, 2015) and Coronel & Morris give explicit attention to these non-relational or NoSQL options. Dennis *et al.* recognize three types: key-value stores, document stores and columnar stores. Key-value stores are databases which function like lookup tables where values to be looked up are indexed by a key as indicated by Figure 1. As the Wikipedia page on key-value stores shows, quite a few implementations are available these days. One of the better-known ones is the open-source implementation Riak. Columnar (or column-oriented) databases are databases which transpose the rows and columns of relational databases. Rows become columns and *vice versa*. This type of representation is meant to imply faster processing for what in the relational world would be column-oriented operations, which in the columnar database become row-oriented operations. Wikipedia contains a list of implementations of this type of database.

Which brings us to the document store, a type of NoSQL database which is most relevant for us because that is what TE 2.0 uses. Document stores, aka Document Management Systems, have quite

a history. Already in the 1980s several systems were commercially available to manage an organization's paper documents, followed by systems for managing electronic documents. However, whereas these systems typically enforced their own binary representation on documents or stored the documents in their native formats and then kept their metadata for sorting and searching, many modern NoSQL document stores store documents as text in a standard format such as XML or JSON. RavenDB and MongoDB, for example, store documents in JSON format. The fact that there typically are no constraints on the data other than these text-formatting ones is important because it implies that as long as a text complies with the format, it is considered a valid 'document' regardless of its information content. Hence, from the document store's perspective, the JSON text `{"foo": "bar"}` is as much a 'document' as a complex JSON structure representing a TeachEngineering lesson.

This notion of a document as a structured text shares characteristics with the notion of classes and objects in object-oriented programming (OOP) but also with tables in a relational database. But whereas OOP objects typically live in program memory and are stored in CPU-specific or byte-compiled binary formats, NoSQL documents exist as text. And whereas in relational databases we often distribute the information of an entity or object over multiple tables in order to satisfy normal form, in NoSQL document stores —as in key-value stores— we often duplicate data and neither maintain nor worry about the database integrity constraints so familiar from relational databases.

So, What Gives?

One might ask why these NoSQL data stores came to the fore anyway? What was so wrong with the relational model that made

NoSQL alternatives attractive? And if the NoSQL databases are indeed 'no SQL,'[1] how does one interact with them?

Let us first say that on the issue of which of these alternatives is better, the dust has by no means settled. The web is rife with articles, statements and blog posts which offer some angle, often highly technical, to sway the argument one way or the other. Yet a few generally accepted assessments can be made (refer to Buckler (2015) for a similar comparison).

- There are few if any functions which can be fulfilled by one that cannot be fulfilled by the other. Most if not everything that can be accomplished with a NoSQL database can be accomplished with a relational database and *vice versa*. However…
- Modern, industry-strength relational databases are very good at providing so-called 'data consistency;' *i.e.*, the property that everyone looking at the data sees the same thing. Whereas such consistency is particularly important in transaction processing —a bank balance should not be different when looked at by different actors from different locations, consistency is less important in non-transaction data processing. A good example would be pattern-finding applications such as those used in business intelligence or business analytics. A pattern is not stable—and hence, not a real pattern—if a single observation changes it. For example, when analyzing the 30-day log of our web server, any single request taken from the log should not make a difference. And if it does, we did not have a stable pattern anyway.
  NoSQL databases are not always equipped with the same consistency-enforcing mechanisms as the relational ones such as locking and concurrency control. For example, in the NoSQL

1. Some interpret the term NoSQL as 'No SQL.' Others interpret it as 'Not Only SQL.'

database RavenDB, operations on individual documents are immediately consistent. However, queries in RavenDB at best guarantee '[eventual consistency](#);' *i.e.*, the notion that, on average or eventually, different observers see the same thing. Although eventual consistency is insufficient for transaction processing, it is often sufficient in non-transactional contexts.

- Because unlike relational databases NoSQL databases are not encumbered with 40 years of engineering investments, their licenses are a lot less expensive, especially in cases where lots of data must be distributed over several or lots of different machines and different locations. Whereas that does not help when consistency is needed, it can weigh significantly if eventual consistency is good enough.
- Relational databases enforce relational integrity (read: primary and foreign key constraints). NoSQL databases do not. But what if we are willing to sacrifice some automatic integrity checking for simpler application code or faster running programs? As we mentioned in the previous chapter, the TE 2.0 JSON representation of documents contains a lot(!) of duplicated data when compared with the TE 1.0 representation, and yet, the 2.0 system runs faster and the codes are less complicated. Of course, making a correction to all that duplicated data would require more work, but if these corrections are rare and can be accomplished without service interruption…
- NoSQL databases tend to be light weight; *i.e.*, no complicated and distributed client-server systems must be set up (hence, their cost advantage). However, this does not imply that complicated systems cannot be built with these systems. On the contrary, NoSQL databases are often used in highly distributed systems with multiple levels of data duplication, and complex algorithms must often be devised to retrieve information from these various locations to then be locally aggregated.
- Relational databases have predefined schemas meaning that

only specific types of data can be stored in predefined tables. NoSQL databases do not have this constraint. Referring to the JSON databases such as RavenDB and MongoDB again, one is free to store the `{"foo": "bar"}` document alongside a complex TeachEngineering lesson. One can easily conceive of some (dis)advantages either way.
- So-called joins; *i.e.*, querying data across multiple tables in a single SQL query, are the bread and butter of relational databases. NoSQL databases do not have joins. Of course, not having joins while having a database in normal form would mean a lot of extra programming, but since in a noSQL database we are more than welcome to ignore normal form, not having joins does not have to be a problem.[2]
- Relational databases traditionally scale vertically by just adding more CPU's, memory and storage space. NoSQL databases scale horizontally by adding more machines.
- Querying. Whereas SQL is the universal language for interacting with relational databases, no such language is available for NoSQL databases. Since NoSQL is not a standard, there is no standard querying protocol. Therefore, most NoSQL implementations have their own syntax and API. However, because of SQL's installed base and popularity, some NoSQL databases offer SQL-like querying protocols. Moreover, since many software applications –databases or not– offer some sort of data retrieval mechanism, efforts to develop supra or '*über*' query languages are underway. We have already seen one of these in the previous chapter, namely Yahoo Query

2. Although standard instructional texts on relational database design emphasize design for normal form, practitioners frequently denormalize (parts of) a relational database for the same reason; namely to write easier and/or faster code.

> Language (YQL), a SQL-like language which can be used to query across Yahoo services. Similarly, Microsoft has developed its Language INtegrated Query (LINQ), a SQL-like language for programmatic querying across its .NET products.

We also must stress that as relational and NoSQL technologies evolve, the lines between them are blurring. For instance, PostgreSQL, a mature and popular open-source relational database supports JSON as a first-class data type. This affords developers the option of using both relational and NoSQL patterns where appropriate in the same application without needing to select two separate database management systems.

TE 1.0: XML/Relational

In this section we discuss the XML/relational architecture used in TE 1.0. If you just want to learn about the TE 2.0 JSON/NoSQL version, feel free to skip this section. However, this section contains some concepts and ideas which you might find interesting, and it assists a better understanding of the relational-NoSQL distinctions.

When we designed TE 1.0 in 2002 NoSQL databases were just a research topic, and using a relational database as our main facility for storing and retrieving data was a pretty much a forgone conclusion. What was nice at the time too was the availability of MySQL; a freely available and open-source relational database.

We had also decided to use XML as the means for storing TeachEngineering content; not so much for fast querying and searching, but as a means to specify document structure and to manage document validity (refer to Chapter 3 for details on XML and validity checking). Consequently, we needed a way to index the content of the documents as written by the curriculum authors into the MySQL relational database.

At the time that we were designing this indexing mechanism,

however, the structure of the TeachEngineering documents was still quite fluid. Although we had settled on the main document types (curricular units, lessons and activities) and most of their mandatory components such as title, summary, standard alignments, *etc.*, we fully expected that components would change or be added in the first few years of operations. Moreover, we considered it quite likely that in the future entirely new document types might have to be added to the collection.[3] As mentioned above, however, relational databases follow a schema and once that schema is implemented in a table structure and their associated integrity constraints set up, and once tables fill with data and application codes are written, database schema changes become quite burdensome. This then created a problem. On the one hand we knew that a relational database would work fine for storing and retrieving document data, yet the documents' structure would remain subject to changes in the foreseeable future and relational databases are not very flexible in accommodating these changes.

After some consideration, it was decided to implement an auto-generating database schema; *i.e.*, implement a basic database schema which, augmented with some basic application code, could auto-generate the full schema (Reitsma *et al.*, 2005). With this auto-generation concept we mean that database construction; *i.e.*, the actual generation of the tables, integrity constraints and indexes occurs in two steps:

- The first step consists of a traditional, hardwired schema of meta (master) tables which are populated with instructions on how the actual document database must be constructed.
- In a second step, a program extracts the instructions from the

3. Although much later than originally expected, in 2014 a brand-new document type, the so-called 'sprinkle' was indeed added.

meta tables, builds the data tables accordingly and then processes the documents, one by one, to index them into those data tables.

This approach, although a little tricky to set up, has the advantage that the entire document database schema other than a small set of never-changing meta tables, is implemented by a program and requires no human intervention; *i.e.*, no SQL scripts for schema generation have to be written, modified or run. Better still, when structural changes to the database are needed, all we have to do is change a few entries in the meta tables and let the program generate a brand-new database while the existing production database and system remain operational. When the application codes that rely on the new database structure are ready, just release both those codes and the new database and business continues uninterrupted with a new database and indexing structure in place.

Data Tables

The following discusses a few of the data tables in the TE 1.0 database:

- Although the various document types have certain characteristics in common; *e.g.*, they all have a title and a summary, the differences between them gave sufficient reason to set up document-type specific data tables. Hence, we have a table which stores activity data, one which stores *lesson* data, one which stores *sprinkle* data, *etc*. However, since we frequently must retrieve data across document types; *e.g.*, '*list all documents with target grade level x*,' it can be beneficial to have a copy of the data common to these document types in its own table. This, of course, implies violating normal form, but such denormalization can pay nice dividends in programming

productivity as well as code execution speed.

- K-12 standards have a table of their own.
- Since the relationship between TeachEngineering documents and K-12 educational standards is a many-to-many one, a document-standard associative table is used to store those relationships. Foreign key constraints point to columns in the referenced tables.
- Since the TeachEngineering collection consists of several hierarchies of documents; for instance, a curricular unit document referring to several lessons and each lesson referring to several activities (see Chapter 1), we must keep track of this hierarchy if we want to answer questions such as '*list all the lessons and activities of unit x*.' Hence, we keep a table which stores these parent-child relationships.
- TeachEngineering documents contain links to other TeachEngineering documents and support materials as well as links to other pages on the web. In order to keep track of what we are pointing to and of the status of those links, we store all of these document-link relationships in a separate table.
- Several auxiliary tables for keeping track of registered TeachEngineering users, curriculum reviews, keywords and vocabulary terms, *etc*. are kept as well.

Meta tables

To facilitate automated schema generation, two meta tables, *Relation* and *Types* were defined.

***Relation* table (definition)**				
Field	**Type**	**Null**	**Key**	**Default**
id	int(10) unsigned	NO	PRI	NULL
groupname	varchar(100)	NO		
component	varchar(100)	NO		

***Relation* table (sample records)**		
id	**Groupname**	**component**
22	Activity	cost_amount
6	Activity	edu_std
72	Activity	engineering_connection
18	Activity	grade_target
21	Activity	keywords
5	Activity	summary
70	Activity	time_in_minutes
4	Activity	title
52	child_document	link_text
50	child_document	link_type
49	child_document	link_url
46	Vocabulary	vocab_definition
45	Vocabulary	vocab_word

***Types* table (definition)**				
Field	**Type**	**Null**	**Key**	**Default**
id	int(10) unsigned	NO	PRI	NULL
name	varchar(100)	NO		
expression	varchar(250)	NO		
cast	enum('string','number','group','root')	NO		string
nullable	enum('yes''no')	YES	YES	NULL
datatype	varchar(50)	YES		NULL

Types table (sample records)				
id	**name**	**expression**	**cast**	**nullab**
19	child_document	/child_documents/link	group	NULL
24	cost_amount	/activity_cost/@amount	number	yes
25	cost_unit	/activity_cost/@unit	string	yes
6	edu_std	/edu_standards/edu_standard	group	NULL
1	edu_std_id	/@identifier	string	yes
33	engineering_category	/engineering_category_TYPE/@category	string	yes
14	grade_lowerbound	/grade/@lowerbound	number	yes
13	grade_target	/grade/@target	number	no
15	grade_upperbound	/grade/@upperbound	number	yes
26	keywords	/keywords/keyword	group	NULL
27	keyword		string	yes

Records in the *Relation* table declare the nesting (hierarchy) of components. For instance, an *activity* has a *title*, a *summary*, *etc*. Similarly, any reference to a *child document* has a *link_text*, a *link type* and a *link_url*. Note that this information is similar to that contained in the documents' XML Schema (XSD). Essentially, the *Relation* table declares all the needed data tables (*groupname*) and those tables' columns (*component*).

The *Types* table in its turn declares for each component its *datatype*, *nullability* as well as an XPath *expression* to be used to extract its content from the XML document. For example, the cost associated with an activity (*cost_amount*) is a *float*, can be null (*yes*) and can be extracted from the XML document with the XPath expression */activity_cost/@amount*. Similarly, the *grade_target* of a document is an *int*(10), cannot be null (*no*) and can be extracted with the XPath expression */grade/@target*. Note that array-like datatypes such as *edu_standards* are not a column in a table. Instead, they represent a list of individual standards just as *keywords*

is a list of individual keywords. They have an XPath expression but no datatype.[4]

Between the *Relation* and the *Types* tables, the entire data schema of the database is declared and as a side effect, for each column in any of the tables, we have the XPath expression to find its document-specific value. Hence, it becomes relatively straightforward to write a program which reads the content from these two tables, formulates and runs the associated SQL `create table` statements to generate the tables, uses the XPath expressions to extract the values to be inserted into these tables from the XML documents, and then uses SQL `insert` statements to populate the tables.

What is particularly nice about this approach is that all that is needed to integrate a brand new document type into the collection is to add a few rows to the *Types* and *Relation* tables. The auto-generation process takes care of the rest. Hence, when in 2014 a new so-called *sprinkle* document type was introduced –essentially an abbreviated activity– all that had to be done was to add the following rows to the *Types* and *Relation* tables:

4. XPath is a language for extracting content from XML pages.

Records added to the *Relation* table to store sprinkle document data		
id	**groupname**	**component**
84	sprinkle	Title
86	sprinkle	total_time
87	sprinkle	sprinkle_groupsize
88	sprinkle	total_time_unit
89	sprinkle	grade_target
90	sprinkle	grade_lowerbound
91	sprinkle	grade_upperbound
93	sprinkle	sprinkle_cost_amount
96	sprinkle	Link
97	sprinkle	time_in_minutes
98	sprinkle	engineering_connection
103	sprinkle	Summary
104	sprinkle	dependency
105	sprinkle	translation

Records added to the *Types* table for storing sprinkle document data					
id	**name**	**expression**	**Cast**	**nullable**	**datatype**
34	sprinkle	/sprinkle	Root	NULL	NULL
35	sprinkle_groupsize	/sprinkle_groupsize	Number	yes	int(10)
36	sprinkle_cost_amount	/sprinkle_cost/@amount	Number	yes	float

Adding these few records resulted in the automatic generation of a s*prinkle* table with the requisite columns, their declared datatypes and nullabilities. Extraction of sprinkle information from the sprinkle XML documents to be stored in the tables was done automatically through the associated XPath expressions. Not a single manual SQL `create table` or `alter table` statement had to be issued, and no changes to the program which generates and populates the database had to be made.

TE 2.0: JSON/NoSQL

The process described in the previous section worked fine for almost 13 years during which time it accommodated numerous changes to the documents' structure such as the addition of the *sprinkle* document type. During those years the collection grew from a few hundred to over 1,500 documents. Yet when the decision was made to rebuild the system, the architectural choice of having a separate store of XML-based documents to be indexed into a relational database was questioned in light of the newly available NoSQL document stores. Why not, instead of having two more or less independent representations of the documents (XML and relational database), have just one, namely the document database; *i.e.*, a database which houses the documents themselves. If that database of documents can be flexibly and efficiently searched, it would eliminate a lot of backend software needed to keep the document repository and the database in sync with each other. Better still, when rendering documents in a web browser one would not have to retrieve data from both sources and stitch it all together anymore. Instead, it could just come from a single source.

To illustrate the latter point consider the way a document was rendered in TE 1.0. Figure 2 shows a section of an activity on heat flow. In the *Related Curriculum* box the activity lists its parent, the *Visual Art and Writing* lesson. This 'parental' information, however, is not stored on the activity itself because in TeachEngineering, documents declare their descendants but not their parents. In TE 1.0 we rendered this same information; *i.e.*, the lesson from which the activity descended. However, whereas in TE. 1.0 most of the rendered information came directly from the XML document, the document's parent-child information was retrieved from the database. Hence, two independent sources of information had to be independently queried and retrieved, each across networks and different computers, only to then be stitched together into a single HTML Web page.

We could have, in TE 1.0, stored not just the typical lookup information of documents in the database; *e.g.*, title, target grade, required time, summary, keywords, *etc.*, but also the entire text of documents. Had we done that we would have only had to access a single source of information for rendering. Except… we did not. In hindsight, perhaps, we should have?

Fast forward to 2015, TE 2.0 and the availability of NoSQL document databases such as MongoDB and RavenDB. Now, we no longer have to separate document content from document searching since the basic data of these databases are the documents themselves. Hence, we have everything we want to know about these documents in a single store. In addition, these databases, partly because their data structures are so simple (no multi-table normalized data structures and no referential constraints), are really fast!

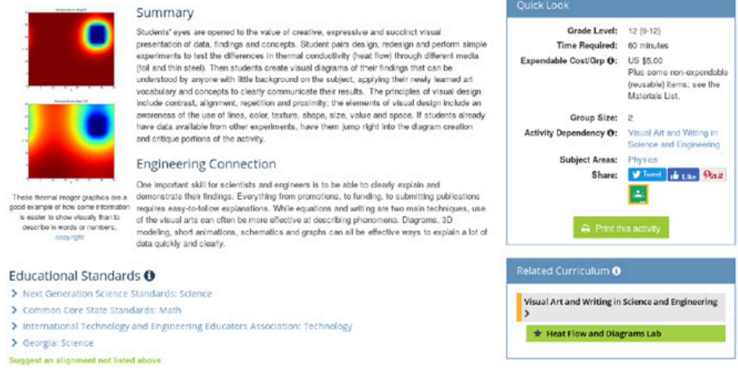


Figure 2: *Heat flow activity lists Related Curriculum*

Earlier in this chapter, we discussed how RavenDB provides eventual consistency for document queries. Clearly, consistency is something that requires careful consideration when designing an application. In a highly transactional application, receiving out-of-date data from queries could be quite problematic. For instance, a

query on a bank account or credit card balance or on the number of available seats on a future airplane ride must be correct at all times. But for a system such as TeachEngineering, eventual consistency is just fine. Curriculum documents do not change that often, and even if they do, it is perfectly acceptable if queries return a previous version for a (limited) period of time. Similarly, when a new document is made available it would be perfectly acceptable if that document is not immediately available everywhere in the world. Moreover, due to the relatively small number of documents stored in TE 2.0's RavenDB database, the' eventual' in 'eventual consistency' means in practice that queries return up-to-date results in a matter of seconds after a change to is made.

Some Practice with two JSON Document Stores: RavenDB and MongoDB

In the remainder of this chapter we work through some practice examples running against two well-known JSON document stores: *RavenDB* and *MongoDB*. We will run the same examples against both databases so that you acquire a feeling how interacting with them is different, yet how the concepts behind them in terms of non-relational JSON storage are quite similar if not identical. For both document stores we first use a very simple example of four very simple JSON documents. The example (*swatches*) is taken from MongoDB's documentation pages. Next, we run a more realistic (but still very small) example of six TeachEngineering so-called 'sprinkle' documents. Sprinkles are abbreviated versions of TeachEngineering activities.

Exercise 5.1: A Little RavenDB

One of the freely available JSON document databases is RavenDB. RavenDB was written for .NET. At the time of this writing RavenDB 4 is available. It supports multiple platforms/operating systems.

We will install a Windows version of RavenDB locally and communicate with it using C#.

- Download the RavenDB installer from https://ravendb.net/downloads.
- Unzip the *RavenDB-4....zip* file to an appropriate place on your system; *e.g.*, *c:\temp*.
- Right-click the file *run.ps1* and select the *Run with PowerShell* option from the menu.
- You will be prompted from the PowerShell window to change the execution policy. Anwer (type) with 'A'.
- The installation script loads a license agreement page into your browser. Please read it and *Accept* it.
- Another page will be loaded into your browser. It offers a choice between a *secure* and *unsecure* installation. Choose *unsecure*.
- The next page allows you to specify the port on which the Raven DB server will be listening. The default is port 8080. Select *Next*.
- RavenDB will next be installed and its server started.
- We can communicate with the server in several ways. We will use HTTP at port 8080 on the local

machine: *http://127.0.0.1:8080*.

For our example, take a look at the following JSON array containing four *swatch* documents[5]. Each document represents a fabric swatch with the following characteristics:

- *name*
- *qty* (quantity)
- *tags*, each having:
    - string describing the material; *e.g.*, *cotton* or *wool*
    - *size* containing:
        - *h* (height)
        - *w* (width)
        - *uom* (unit of measurement)

```
[
{ "name": "cotton_swatch", "qty": 100, "tags":
  ["cotton"], "size": { "h": 28, "w": 35.5,
  "uom": "cm" } },
{ "name ": "wool_swatch", "qty": 200, "tags":
  ["wool"], "size": { "h": 28, "w": 35.5,
  "uom": "cm" } },
{ "mame ": "linen_swatch", "qty": 300, "tags":
  ["linen"], "size": { "h": 28, "w": 35.5,
  "uom": "cm" } },
```

5. Swatch: a sample or example item; typically a sample piece of fabric or other type of material.

```
{ "name ": "cotton_swatch", "qty": 100, "tags":
  ["cotton"], "size": { "h": 50, "w": 50,
  "uom": "cm" } }
]
```

These are the four ‘documents’ which we will programmatically store and query in RavenDB.

We will be using Visual Studio and the C# language.[67] Our program will communicate with RavenDB over HTTP.

For this example we will be using the standard (de)serialization approach; *i.e.*, we will hold the documents in objects of the `Swatch` class in our C# program and then programmatically store them in RavenDB. RavenDB will store them as JSON strings (documents). After they are stored we will run a few queries against the database.

In order to hold the swatches as objects, we need a class definition for them. We can, of course, figure this out by ourselves, but let us use the services of others and have

6. Visual Studio is Microsoft’s .NET Integrated Development Environment (IDE). Although it is available in several configurations, for these examples we will be using the freely available Community Edition.
7. There is a public RavenDB playground available for testing and experimentation. It can be accessed at http://live-test.ravendb.net/. For more information, visit https://ravendb.net/docs/article-page/3.5/csharp/start/playground-server.

http://json2csharp.com/ do it for us. Simply enter the above JSON array in and click *Generate*. The result:

```
  public class Size
{
   //we will use a 'decimal' instead
   public int h { get; set; }
   // we will use a 'decimal' instead
   public double w { get; set; }
   public string uom { get; set; }
}

public class RootObject //we will use 'Swatch' instead
{
   public string name { get; set; }
   public int qty { get; set; }
   public List<string> tags { get; set; }
   public Size size { get; set; }
}
```

With these class definitions we can write our Raven program. However, in order for our program to interact with RavenDB, we must first install the required *RavenDB Client* on our machine:

- *Project –> Manage NuGet Packages…*
- Find *RavenDB.Client*
- Click *Install*
- Click OK

Here is the (console) C# program:

```
  using System;
using System.Collections.Generic;
```

```
using System.Linq;
using Raven.Client.Documents;
using Raven.Client.Documents.Indexes;
using Raven.Client.Documents.Operations;
using Raven.Client.Exceptions;
using Raven.Client.Exceptions.Database;
using Raven.Client.ServerWide;
using Raven.Client.ServerWide.Operations;

  namespace RavenDB
{
    public class Swatch //Define the Swatch class
    {
        public string Id { get; set; }
        public string Name { get; set; }
        public int Qty { get; set; }
        public List<string> Tags { get; set; }
        public Size Size { get; set; }
    }

    //Define the Size class(every Swatch has a Size)
    public class Size
    {
        public decimal W { get; set; }
        public decimal H { get; set; }
        public string Uom { get; set; }
    }

      public class SwatchIndex :
                  AbstractIndexCreationTask<Swatch>
    {
        /* Method for creating the index. The index is
```

```
            RavenDB's catalog of items it uses to conduct
            retrievals/searches */
         public SwatchIndex()
         {
             Map = items => from item in items
                            select new
                            {
                                item.Name,
                                item.Qty,
                                item.Tags,
                                item.Size.H,
                                item.Size.W,
                                item.Size.Uom
                            };
         }
     }


  //Program which interacts with RavenDB
class Program
{
  //inputs or retrieves JSON data from Raven
  static void Main(string[] args)
  {
    //Create a list of Swatches
    var Items = new List<Swatch>
    {
      new Swatch {
        Id = "items/1", Name = "cotton_item",
        Qty = 100, Tags = new List<string> { "cotton" },
        Size = new Size { H = 28, W = 35.5m, Uom = "cm"}
      },
```

```
        new Swatch {
          Id = "items/2", Name = "wool_item",
          Qty = 200, Tags = new List<string> { "wool" },
          Size = new Size { H = 28, W = 35.5m, Uom = "cm" }
        },
        new Swatch {
          Id = "items/3", Name = "linen_item",
          Qty = 300, Tags = new List<string> { "linen" },
          Size = new Size { H = 28, W = 35.5m, Uom = "cm" }
        },
        new Swatch {
          Id = "items/4", Name = "cotton_item",
          Qty = 100, Tags = new List<string> { "cotton" },
          Size = new Size { H = 50, W = 50, Uom = "cm" }
        }
    };

string databaseName = "foo";

/* We will communicate with Raven over HTTP at
   port 127.0.0.1:8080.
   CHANGE THIS IN THE LINE BELOW DEPENDING ON THE
   PORT ON WHICH RAVENDB IS LISTENING. */
DocumentStore store = new DocumentStore
{
  Urls = new[] { "http://localhost:8080" },
  Database = databaseName
};

store.Initialize();

//Create the database if it doesn't already exist
```

```
CreateDatabaseIfNotExists(store, databaseName);

  //Run index creation
new SwatchIndex().Execute(store);

//Open the store's session
using (var session = store.OpenSession())
{
  //Loop through the Items
  foreach (var item in Items)
  {
    //Store the item in the session
    session.Store(item);
  }
  //Save all pending changes
  session.SaveChanges();

  /* Query for all swatches named "cotton_item"
     and store them in a List */
  var cottonSwatches = session
      .Query<Swatch, SwatchIndex>()
      .Where(x => x.Name == "cotton_item")
      .ToList();

  Console.WriteLine(
  "Swatches with name 'cotton_item':");
  //Write them out
  foreach (var swatch in cottonSwatches)
  {
    Console.WriteLine(swatch.Id);
  }
```

```
      /* Query for all swatches with a Quantity > 100
       and store them in a List */
    var highQuantitySwatches = session
        .Query<Swatch, SwatchIndex>()
        .Where(x => x.Qty > 100)
        .ToList();

    Console.WriteLine("Swatches with quantity > 100:");
    //Write them out
    foreach (var swatch in highQuantitySwatches)
    {
      Console.WriteLine(swatch.Id);
    }
    Console.WriteLine("Hit a key");
    Console.ReadKey();
  }
  }

    //Creates the database if it does not exist
  public static void CreateDatabaseIfNotExists(
         DocumentStore store, string database)
  {
    database = store.Database;
    try
    {
      store.Maintenance.ForDatabase(database).
            Send(new GetStatisticsOperation());
      // The database already exists
    }
    catch (DatabaseDoesNotExistException)
    {
      try
```

```
    {
      //we do not have the database yet; create it
      store.Maintenance.Server.
Send(new CreateDatabaseOperation(
         new DatabaseRecord(database)));
    }
    catch (ConcurrencyException)
    {
      System.Console.WriteLine(
        "Error... creating database");
      System.Environment.Exit(1);
    }
}
}

  }
}
```

So how does all this work?

- We first define the classes `Swatch` and `Size` in accordance with the class structures returned from http://json2csharp.com/.
- We then define an index for swatches: `SwatchIndex`. In RavenDB an index specifies the associations of the properties of documents with those documents so that at some point in the future we can query them. For instance, if we ever want to find documents that have a certain *Name* or *Qty*, we must build an index for those properties. Internally, RavenDB uses Lucene.Net, an open-source search engine library, to do its indexing and querying.
- We then load up a list of `Swatch` objects (called

Items).

- Each of the Swatches in the Items list is then sent to RavenDB for storage.
  - Note that we do not(!) send the Swatches as JSON structures to RavenDB. We could do that too, but in this case we let RavenDB figure out how to make JSON structures from the Swatch objects and store them.

```
  foreach (var item in Items) //Loop through the It
{
  //Store the item in the session
  session.Store(item);
}
```

- After storing the Swatches in RavenDB, we formulate a few queries and send them to RavenDB for retrieval. The results are printed to the console.

  The queries take a form which may look a little odd:

```
var cottonSwatches = session
 .Query<Swatch, ItemIndex>()
 .Where(x => x.name == "cotton_item")
 .ToList();
```

and

```
var highQuantitySwatches = session
 .Query<Swatch, ItemIndex>()
 .Where(x => x.qty > 100)
 .ToList();
```

These queries use RavenDB's *Query*() method,

> which supports .NET's *Language INtegrated Query* or *LINQ*. LINQ has a conceptual similarity with SQL (*select … from … where …*), but it is, of course, not at all relational. Instead, it is a general-purpose language for querying a great variety of data structures. In Microsoft's own words: "*.NET Language-Integrated Query defines a set of general purpose standard query operators that allow traversal, filter, and projection operations to be expressed in a direct yet declarative way in any .NET-based programming language.*"

Note, by the way, that when you run this program several times in a row, the number of swatches stored in RavenDB remains the same. This is because we explicitly specify a value for the *Id* property of each swatch. By convention, RavenDB treats the *Id* property as the unique identifier for each document (similar to a primary key in a relational database table). Hence, each time you run the program the existing documents are overwritten. Alternatively, if the *Id* property was not specified for each document, RavenDB would automatically generate one. Each subsequent run of the program would then add new versions of the documents to the database, each with a unique value for the *Id* property.

Exercise 5.2: A Little MongoDB

Another NoSQL/JSON document database we can practice with is the open source NoSQL database MongoDB.

We will install the MongoDB 4.0.4 (Community) Server locally. We assume a Windows machine, but installs for other platforms are available as well.

- Download the MongoDB Server (as MSI file) from https://www.mongodb.com/download-center?jmp=nav#community
  - Run the downloaded installation script (Select the *Complete* (not the *Custom*) version and uncheck the *Install Compass* option).
- Find the folder/directory where MongoDB has been installed (typically *c:\Program Files\MongoDB*) and navigate to its *\Server\ version_no\bin* folder where the executables *mongod.exe* and *mongo.exe* are stored.
  - *mongod.exe* starts the server instance.
  - *mongo.exe* starts a mongo command shell through which we can send commands to *mongod*.
- By default, MongoDB relies on being able to store data in the *c:\data\db* folder. Either create this folder using the Windows File Browser, or pop up a

Windows command line and run the command:

```
mkdir c:\data\db
```

- If you have not already done so, run *mongod.exe* to start the MongoDB server.

We will communicate with MongoDB in two modes: first, by sending it commands directly using its command shell and then programmatically using C#.

Exercise 5.3: MongoDB Command Shell

- Run *mongo.exe* to start a MongoDB command shell. Enter all commands mentioned below in this command shell).

As in RavenDB, MongoDB structures its contents hierarchically in databases, collections within databases and records (called 'documents') within collections.

- Start a new database *foo* and make it the current database:

```
use foo
```

- Create a collection *fooCollection* in database *foo*:

```
db.createCollection("fooCollection")
```

Now we have a collection, we can add documents to it. We will use the *swatches* documents again (they came directly from [MongoDB's own documentation pages]).

- Add four documents to *fooCollection:*

```
db.fooCollection.insertOne( { "swatch": "cotton_ swatch",
  "qty": 100, "tags": ["cotton"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } })
db.fooCollection.insertOne( { "swatch": "wool_ swatch",
  "qty": 200, "tags": ["wool"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } })
db.fooCollection.insertOne( { "swatch": "linen_ swatch",
  "qty": 300, "tags": ["linen"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } })
db.fooCollection.insertOne( { "swatch": "cotton_ swatch",
  "qty": 100, "tags": ["cotton"],
  "size": { "h": 50, "w": 50, "uom": "cm" } })
```

Now let us start querying *foo.fooCollection*

- Let us see what we have in the collection. Since we pass no criteria to the *find()* method, all documents in *fooCollection* are returned:

```
db.fooCollection.find()
```

- Which *swatches* made of *cotton* do we have?

```
db.fooCollection.find( { "tags": "cotton" } )
```

- Of which *swatches* do we have more than 100?

```
db.fooCollection.find( { "qty": { $gt: 100 } } )
```

Exercise 5.4: MongoDB Programmatic Interaction

Now that we have played with command-driven MongoDB, we can try our hand at having a program issue the commands for us. We will use C# as our language, but MongoDB can be programmatically accessed with other languages as well.

A note before we start. When you google for C# examples of MongoDB interactions, you will find a lot of code with the *async* and *await* keywords. These codes are typically written for applications which will asynchronously process MongoDB queries. With 'asynchronous' we mean that these applications will not sequentially queue up commands, executing them in order and waiting for one to come back before the next one is sent. Instead, they send commands to MongoDB whenever they need to and process the replies in any order in which they come back. This approach makes a lot of sense in production environments where lots of queries must be issued and different queries demand different amounts of time to complete. However, for our simple examples, we will just use more traditional synchronous communication. We issue a single command and wait until it returns before we send the next one.

Instead of hardcoding the JSON documents, we will pick them up from a file. Store the following JSON in the file *swatches.json:*

```
  [
{ "swatch": "cotton_ swatch", "qty": 100,
  "tags": ["cotton"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } },
{ "swatch": "wool_ swatch", "qty": 200,
  "tags": ["wool"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } },
{ "swatch": "linen_swatch", "qty": 300,
  "tags": ["linen"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } },
{ "swatch": "cotton_swatch", "qty": 100,
  "tags": ["cotton"],
  "size": { "h": 50, "w": 50, "uom": "cm" } }
]
```

To validate the syntax of the above JSON segment, check it with an on-line JSON parser such as http://jsonparseronline.com.

We once again will be using Visual Studio and a console app to run these examples. However, in order for our program to interact with MongoDB, we must first install the required *MongoDB* and *Json.Net* packages:

- *Project ->*
- *Manage NuGet Packages*
- Find and select *Newtonsoft.Json*, *MongoDB.Driver* and *MongoDB.Bson*
- Click *Install*.
- Click OK.

Here is the (console) C# program for inserting the four documents stored in the *swatches.json* file into the `swatchesCollection`:

```
  using System;
using System.IO;
using MongoDB.Bson;
using MongoDB.Driver;
using Newtonsoft.Json.Linq;

namespace mongodb_c_sharp
{
  class Program
  {
    /* This code assumes that the swatches.json file is
    stored at C:\swatches.json. If you stored it elsewhere,
    just change the jsonFilePath string below */
    const string jsonFilePath = "C:\\swatches.json";

    // We will use the "swatches" database
    const string dbName = "swatches";

    // We will use the "swatchesCollection" collection
    const string collectionName = "swatchesCollection";

    static void Main(string[] args)
    {
      /* Open a connection to mongoDB.
      We are assuming localhost */
      string connectString = "mongodb://localhost";

      MongoClient client = null;
      try { client = new MongoClient(connectString); }
      catch
      {
        Console.WriteLine("Error connecting to MongoDB. Is m
```

```
        Environment.Exit(1); // Exit on failed connection
      }

      /* Attach to the swatches database (create it if it
      does not yet exist) */
      IMongoDatabase database = client.GetDatabase(dbName);

      // Get the swatchesCollection collection
      var collection = database.GetCollection<BsonDocument>
      (collectionName);

      // Read the JSON file into a string
      StreamReader reader = new StreamReader(jsonFilePath);
      string json_str = reader.ReadToEnd();

      // Load the JSON into a JArray
      JArray arr = JArray.Parse(json_str);

      /* Loop through the JArray and insert each of its
      documents into the database */
      foreach (JObject obj in arr)
      {
        BsonDocument doc =
          BsonDocument.Parse(obj.ToString());
        collection.InsertOne(doc);
      } // end of foreach

    } // end of Main()
  } // end of Program class
} // end of Namespace
```

If you still have your MongoDB command shell open (if

not, just start it again –see previous section on how to do this), let us see what the program accomplished:

```
use swatches

db.swatchesCollection.find()
```

Next, we run the program which queries `swatchesCollection` for those swatches of which we have more than 100 (qty > 100). Here is the (console) program:

```
using System;
using MongoDB.Bson;
using MongoDB.Driver;

namespace mongodb_c_sharp
{
  class Program
  {
    // We will use the "swatches" database
    const string dbName = "swatches";

    // We will use the "swatchesCollection" collection
    const string collectionName = "swatchesCollection";
    static void Main(string[] args)
    {
      /* Open a connection to mongoDB.
      We are assuming localhost */
      string connectString = "mongodb://localhost";

      MongoClient client = null;
      try { client = new MongoClient(connectString); }
```

```
        catch
        {
          Console.WriteLine("Error connecting to MongoDB. Is m
          Environment.Exit(1); // Exit on failed connection
        }

        // Attach to the swatches database
        IMongoDatabase database = client.GetDatabase(dbName);

        // Get the swatchesCollection collection
        var collection = database.GetCollection<BsonDocument>
          (collectionName);

        // Find swatches with "qty" > 100
        // First, build a filter
        var filter = Builders<BsonDocument>
          .Filter.Gt("qty", 100);
        // Then apply the filter to querying the collection
        var resultList = collection.Find(filter).ToList();

        // Write the results to the console
        foreach (var result in resultList)
          Console.WriteLine(result);

        Console.WriteLine("Hit a key");
        Console.ReadKey();

      } // end of Main()
    } // end of Program class
  } // end of Namespace
```

Now, while still in Mongo's command shell, query the *foo*

database again to see if its collection of swatches is still available (). Of course it still is, but it nice to see anyway.

Exercise 5.5: MongoDB Programmatic Interaction: TeachEngineering Example – Sprinkles

Now we have run through a basic (*swatches*) example, we can apply the same approach to a small sample of more complex but conceptually identical TE 2.0 JSON documents. *Sprinkles* are abbreviated versions of TeachEngineering activities. Our sample contains six TE 2.0 *sprinkle* documents and is stored at *http://faculty.bus.oregonstate.edu/reitsma/sprinkles.json*.

Download the content of *http://faculty.bus.oregonstate.edu/reitsma/sprinkles.json* and store it in a file called *sprinkles.json*.[8] Please take a

8. We could, of course, write code which retrieves the JSON over HTTP (from the given URL) rather than first storing the JSON on the local file system. However, for the sake of being able to use pretty much the exact same code as in the previous examples, we will read the JSON from a local file and insert it into the database.

moment to study the content of the file. Notice that it is a JSON array containing six elements; *i.e.*, six sprinkles. Also notice that although the elements are all sprinkle documents and they all have the same structure, their elements are not always specified in the same order. For instance, whereas the third and following documents start with a *Header* element followed by a *Dependencies* element, the first two documents have their *Header* specified elsewhere.

First, just like in the *swatches* case, we will programmatically import the data. We use the exact same program as we used for loading the *swatches* data, except for some changes to the following constants:

- Set the `const dbName` to `sprinkles`.[9]
- Set the `const collectionName` to `sprinklesCollection`.
- Set the `const jsonFilePath` to `c:\\sprinkles.json`.

Assuming that your MongoDB server (*mongod.exe*) is running, running the program will load the six sprinkles into the *sprinkleCollection*.

Now the reverse: programmatically finding the number of sprinkles for which the required time >= 50 minutes (*Time.TotalMinutes* >= 50). Again, we use pretty much the

9. We can, of course, use the swatches database to store sprinkle data, but in order to keep the examples independent of each other, we use a new database.

exact same program as we used for retrieving *swatches* data, except for a few simple changes:

- Set the `const dbName` to `sprinkles`.
- Set the `const collectionName` to `sprinklesCollection`.
- Change the line which defines the query filter as follows:

```
var filter = Builders<BsonDocument>.Filter
 .Gte("Time.TotalMinutes", 50);
```

- Remove (comment out) the `foreach()` loop and replace it with the following:

```
Console.WriteLine(resultList.Count);
```

- The result should be 5 (check it manually against the *sprinkles.json* file).

Exercise 5.6: RavenDB Programmatic Interaction: TeachEngineering Example — Sprinkles

Let us now repeat the sprinkle example we just ran against MongoDb, but this time run it against RavenDB. When you look through the program you will see that unlike what we did in our earlier (swatches) example, this

time we do not generate a search index prior to searching. Instead, we follow the MongoDB example above and search through the documents in real-time without such an index. Of course, in a production environment, generating the indexes ahead of time would save (significant) search time, but for our case here we tried keeping the MongoDB and RavenDB examples similar.

```
  using System;
using System.IO;
// Do not forget to add the RavenDB NuGet package!
using Raven.Client.Document;
using Raven.Json.Linq;

namespace RavenDB
{
  /* Program which interacts with RavenDB using a few
  TeachEngineering 'sprinkle' documents */
  class Program
  {
    //inputs or retrieves JSON data from Raven
    static void Main(string[] args)
    {
      /* This code assumes that the sprinkles.json file is
      stored at C:\sprinkles.json If you stored it
      elsewhere, change the jsonFilePath string below */
      const string jsonFilePath = "C:\\sprinkles.json";

      // Make a new document store
      var store = new DocumentStore
      {
        /* We will communicate with Raven over HTTP at
        localhost:8080 */
```

```
  Url = "http://localhost:8080",
  /* We will communicate with database
  'sprinklesCollection' */
  DefaultDatabase = "sprinklesCollection"
};

store.Initialize(); //Initialize the store

/* This next statement is not appropriate for
production use, but is used to ensure that the
index is up to date before returning query
results */
store.Conventions.DefaultQueryingConsistency =
  ConsistencyOptions.
  AlwaysWaitForNonStaleResultsAsOfLastWrite;

// Read the JSON file into a string
StreamReader reader = new StreamReader(jsonFilePath);
string json_str = reader.ReadToEnd();

// Load the JSON into a JArray
var arr = RavenJArray.Parse(json_str);

/* Loop through the JArray and insert each of its
documents into the database */
foreach (RavenJObject obj in arr)
{
  store.DatabaseCommands.Put(null, null, obj, null);
} // end of foreach

/* Query for sprinkles greater than or equal to 50
minutes in length Note that unlike in the swatches
```

```
        example, we do not create an explicit index ahead
        of time. RavenDB will automatically create an index
        for us. This works for ad-hoc queries, but in a
        typical application it is best to create the indexes
        ahead of time. */
        using (var session = store.OpenSession())
        {
          var results =
          session.Advanced.DocumentQuery<object>()
            .WhereGreaterThanOrEqual("Time.TotalMinutes", 50);
          Console.WriteLine(results.Count());
        }

        Console.WriteLine("Hit a key");
        Console.ReadKey();
      }
    }
  }
```

Once again, running this program should result in a search result of '5.'

Summary and Conclusion

In this chapter we took a look at how TeachEngineering evolved from a system with a relational database at its core and XML representing documents (TE 1.0), to a system running off of a NoSQL database with JSON representing documents (TE 2.0). Whereas the relational/XML model worked fine during the 12 years of TE 1.0, the new NoSQL/JSON alternative provides the advantage that JSON

is used as both the document and the database format. This unification of representation significantly reduces system complexity from a software engineering point of view.

We also argued that whereas the property of ‘consistency’ which is well entrenched in industry-strength relational databases, is of crucial importance in transactional settings, ‘eventual consistency’ is plenty good for an application such as TeachEngineering. As such, we can forgo much of the concurrency control facilities built into relational databases and instead rely on a less advanced but also much less expensive and easier to maintain NoSQL database.

Similarly, since the data footprint for a system such as TeachEngineering is quite small —the current size of the RavenDB database is a mere 170 MB— replicating some data in a controlled and managed way is quite acceptable. This again implies that we can forego the mechanisms for adhering to and maintaining strict normal form, and that in turn implies that we do not need sophisticated data merge and search methods such as relational table joins.

We very much care to state that none of the NoSQL material we have discussed and practiced here deters from the value and utility of relational databases. For systems which require ‘consistency,’ relational databases with their sophisticated built-in concurrency controls continue to be a good and often the best choice. But if ‘eventual consistency’ is good enough and if you have data governance policies and practices in place which prevent the inconsistent representation of data, then you should at least consider a modern NoSQL database as a possible candidate for storing your data.

References

Buckler, C. (2015) SQL vs. NoSQL: The Differences. Available:

https://www.sitepoint.com/sql-vs-nosql-differences/. Accessed: 12/2016.

Codd, E. F. (1970) A Relational Model of Data for Large Shared Data Banks. *Communications of the* ACM. 13. 377-387.

Codd, E. F. (1982) Relational database: A Practical Foundation for Productivity. *Communications of the* ACM. 25. 109-117.

Dennis, A., Wixom, B.H., Roth, R.A. (2014) *System Analysis and Design*. 5th ed. John Wiley & Sons.

Dennis, A., Wixom, B.H., Tegarden, D. (2012) *System Analysis & Design with UML Version* 2.0. John Wiley & Sons.

Dennis, A., Wixom, B.H., Tegarden, D. (2015) *System Analysis & Design. An Object-Oriented Approach with UML. 5th ed.* John Wiley & Sons.

Reitsma, R., Whitehead, B., Suryadevara, V. (2005) Digital Libraries and XML-Relational Data Binding. *Dr. Dobbs Journal*. 371. 42-47.

Satzinger, J., Jackson, R., Burd, S. (2016) *Systems Analysis and Design in a Changing World*. CENGAGE Learning.

Tilley, S., Rosenblatt, H. (2017) *Systems Analysis and Design*. CENGAGE Learning.

Valacich, J.S., George, J.F., Hofer, J.A. (2015) *Essentials of Systems Analysis and Design*. Pearson.

6. Document Accessioning

Introduction

All libraries, digital or not, have processes for formally accepting and including items into their collection; a process known as accessioning, and for removing items from their collection known as deaccessioning. In this chapter we contrast and compare the accessioning methods of TE 1.0 and 2.0. We will see that, once again, the choice of XML *vs*. JSON, although not strictly a cause for difference in accessioning approaches, almost naturally led to differences between the two architectures. The most significant of these differences is that whereas in TE 1.0 document editing and accessioning where two separate processes executed and controlled by different people, in TE 2.0 they became integrated into a single process executed by the document editor.

Authoring ≠ Editing

One way in which we can categorize digital libraries is in content collections and meta collections. In a meta collection, no actual content is kept; only meta data are kept. A good example of a meta data collection is *NSDL.org*. NSDL (or National Science Digital Library) maintains meta data of about 80 digital library collections and allows faceted searches over those 80 collections. The actual items themselves, however, are held by the various collections over which NSDL can search.

For meta collections, accessioning tends to be a relatively simple process, mostly because each item they represent –a so-called meta record– tends not to contain much data. In fact, in many cases this

accessioning is fully or semi-automated in that it can be entirely accommodated with Web services offered by the various libraries which allow their meta data to be collected by the meta collection. Of course, the main difficulty for meta collections is to keep them synchronized with the content collections they reference. Items newly added to the content collections must be referenced, without too long a delay, in the meta collection, and documents no longer available in the content collections must be dereferenced or removed from the meta collection.

For content collections, however, accessioning tends to be more complicated, partly because the items to be accessioned are more complicated and partly because they often have to be reformatted.

TeachEngineering documents —in both TE 1.0 and TE 2.0— are typically submitted by their authors as text-processed documents; most often Microsoft Word documents. Their authors are neither asked nor required to maintain strict formatting rules, but they are required to provide specific types of information for specific types of content such as a summary, a title, grade levels, *etc*. Depending on the type of document, entire text sections are either mandatory or optional. TeachEngineering lessons, for instance, must have a *Background* section and activities must have an *Activity Procedure* section. TeachEngineering document editors work with the authors to rework their documents so that they comply with the required structure. Once done, however, the documents are still in text-processed form. Hence, as we have learned in the previous chapter, the first step of accessioning consists of converting them from text-processed form into the format required by the collection: XML for TE 1.0; JSON for TE 2.0. This conversion is done by special editing staff known internally as 'taggers.'[1]

1. The term *tagger* stems from the TE 1.0 period during which document conversion consisted of embedding content in XML 'tags.'

TE 1.0 Tagging: Word XML

As discussed in the previous chapter, all TE 1.0 documents were stored as XML. Hence, conversion of their content as written by their authors to TE-specific XML was the main objective of the tagging process. This constituted a problem because the TE-XML specification was complex and asking taggers to themselves apply the proper tags to document content would almost certainly lead to failure. Moreover, as mentioned in chapter 3, the TE XML contained both content and some formatting tags. This mixing of tag types and the myriad of validation rules associated with these tags made it essentially impossible for student workers —the TE 1.0 tagging staff consisted mainly of student workers— to directly edit the documents in XML.

Of course, we as TeachEngineering were not the only ones having this problem. With the rapidly increasing popularity of XML came the common need to convert documents from one form or another into XML, and this task is not very human friendly.

Fortunately, Altova, a company specializing in XML technology made available (for free) its *Authentic* tool for in-document, what-you-see-is-what-you-get (WYSIWYG) editing of XML documents. With Authentic, TE taggers could view and edit TE documents without having to know their XML, yet Authentic would save their document in TE-XML format. Moreover, since Authentic kept track of the TE-XML schemas —recall that an XML schema is the specification of the rules of validity for a particular type of XML document— Authentic protects editors from violating the Schema, thereby guaranteeing that documents remain valid.

Figure 1: *Editing a section of the Tug of War activity's XML representation in Altova's Authentic.*

Introduction/Motivation

A tug of war contest is a battle involving many forces
copyright

(In advance, be prepared to show students the 16-slide Tug of War Battle Bots Presentation, a PowerPoint® file.)

Figure 2: *Section of Tug of War activity rendered in TeachEngineering*

Figure 1 shows part of a TE 1.0 activity edit session using Authentic. Note how the look-and-feel of the activity as seen in Authentic is quite different from the look-and-feel of that same activity when rendered in TE 1.0 (Figure 2). There are two reasons for this difference. First and foremost, XML is (mostly) about content and content can be rendered in many different ways. Second, because XML is (mostly) about content, no great effort was neither made nor needed to precisely render the activity in Authentic as it would render in TeachEngineering. Still, in order to show a tagger the

rendered version of the document, the TE 1.0 system offered a (password-protected) w

eb page where the tagger could test render the document.

You might, at this point, be wondering who then determines the look-and-feel of the Authentic version of the document and how that look-and-feel is set up? This would be a good question and it also points out the cleverness of Altova's business model. In a way, Altova's business model associated with Authentic is the reverse of that of Adobe's business model associated with its *PDF Reader*. Adobe gives away PDF Reader as a loss leader so that it can generate revenue from other PDF-generating and PDF-processing products. Demand for these products would be low if few people can read what comes out of them. With Authentic we have the reverse situation. Altova sells tools for generating and validating XML schema's. One of the uses of those schemas is for people to edit documents in XML which follow those schemas. So Altova makes the Authentic XML editor available free of charge but generates revenue with the tools that produce the files –XSDs and Authentic WYSIWIG document layouts– with which documents can be edited in Authentic. Hence, in TE 1.0, the TE engineers used Altova tools to construct the document XSDs and to generate a layout for WISYWIG editing in Authentic. TE taggers then used the free-of-charge Authentic tool to do the actual document editing and used a TeachEngineering test-rendering service to see the rendered version of the edited document.

TE 1.0 Document Ingestion and Rendering

Although XML editing of the document was the most labor-intensive step of the accessioning process, once we had an XML version of a TE document, we were not quite there yet as it still had to be registered to the collection. In TE 1.0 this was done in three steps.

1. **Document check-in**. The tagger would check the document into a central code repository (aka version control) system. Code repository systems such as CVS, Subversion, Team Foundation Version Control, and Git maintain a history and a copy of all code changes, allow reverting to previous versions of the code, track who made which change when, and can checkpoint whole collections of code in so-called code branches or releases. They are indispensable for code development, especially when more than one coder is involved. Although developed for managing program source code, these systems can of course be used for tracking and maintaining other types of electronic data sets as well, for instance XML or JSON documents. Hence, in TE 1.0, taggers, once a document had been converted to XML, checked that document into a central code repository system.
2. **Meta data generation**. Once checked into the code repository system, a program run once or twice a day would extract data from the XML documents and generate meta data for them. Recall from chapter 3 that these meta data were served to third party users interested in TeachEngineering contents. One of those was NSDL.org whose data-harvesting programs would visit the TeachEngineering meta data web service monthly to inquire about the state of the collection. A side effect of this meta data generator, however, was additional quality control of the content of the document. As we have seen, XSDs are an impressive quality control tool as they can be used to check the validity of XML documents. Such validity checking, however, is limited to the syntax of the document. Hence, a perfectly valid XML document can nevertheless have lots of problems. For example, it may contain a link to a non-existing image or document or it may declare a link to a TE lesson whereas in fact the link points to an activity. One of the things that the meta data generator did, therefore, was to conduct another set of quality control checks on the documents. If it deemed a document to be in violation of one

or more of its rules, no meta data would be generated for it and the document would not be ingested into the collection. It would, of course, remain in the code repository system from which the tagger could then check it out, fix the problem(s) flagged by the meta data generator and check it back in for the next round of meta data generation.

3. **Document indexing**. Twice daily TE 1.0 ran a process which would actually ingest newly created or modified documents. We named this process 'the spider' because its method of picking up documents for ingestion was very similar to that of so-called web crawlers, aka 'web spiders.' Such a crawler is a process which extracts from a document all the data it is looking for, after which it then looks for references or links to other documents and then crawls those in turn. Whereas most modern crawlers are multi-threaded; *i.e.*, they simultaneously crawl more than one document, the TE 1.0 spider was simple and processed only one document at the time. This was perfectly acceptable, however, because although the overall process would complete more quickly if crawls would run in parallel, we only had to complete the process once or twice a day. Figure 3 shows the process of spidering a TE 1.0 document; in its generic form as pseudo code (a) and as an example of spidering a TE 1.0 curricular unit on heart valves (b). Note how the process in (a) is recursive, *i.e.*, the *spider*() method contains a call to *spider*().

(a)

```
spider (document)
{
  document.index_content(); // index the document
  doc_links = document.find_links(); // find links in t
                                        // document
  foreach (doc in doc_links) // spider all doc_links
    if (spidered(doc) == false) // only spider when
                                // not yet spidered
    spider(doc);
}
```

(b)

Curricular Unit: Aging Heart Valves:

- *Lesson: Heart to Heart:*
  - *Activity: The Mighty Heart*
  - *Activity: What's with all the pressure?*
- *Lesson: Blood Pressure Basics:*
  - *Activity: Model Heart Valves*

0. ***Spider curricular unit:***
 Aging Heart Valves:
  Index content of Aging Heart Valves
  Doc_links found:
   Lesson: Heart to Heart
   Lesson: Blood Pressure Basics
   1. ***Spider lesson:*** *Heart to Heart:*
    Index content of Blood Pressure Basics
    Doc_links found:
     Activity: The Mighty Heart
     Activity: What's with all the pressure?
     1a. ***Spider activity*** *The Mighty Heart:*
      Index content of The Mighty Heart
      Doc_links found: none
     1b. ***Spider activity*** *What's with all the pressure?:*
      Index content of What's with all the pressure?
      Doc_links found: none
   2. ***Spider lesson:*** *Blood Pressure Basics:*
    Index content of Blood Pressure Basics
    Doc_links found:
     Activity: Model Heart Valves
     2a. ***Spider activity*** *Model Heart Valves:*
      Index content of Model Heart Valves
      Doc_links found: none

Figure 3: TE 1.0 document spidering. Generic algorithm (a) and Curricular Unit examp

4. **Document rendering**. One last step in the document production chain in both TE 1.0 and 2.0 is actual rendering of documents in users' browsers. To a large extent, this is the simplest of the production steps, although it too has its challenges. Rendering in TE 1.0 was accomplished in PHP, then one of the more popular programming languages for Web-based programming.
   Rendering a TE 1.0 document relied partly on information stored in a document's XML content and partly on information stored in the database generated during document indexing. Whereas all of the document's content could be rendered directly from its XML, some aspects of rendering required a database query. An example is a document's 'Related Curriculum' (Figure 4). Whereas a document may have 'children;' *e.g.*, a lesson typically has one or more activities, it does not contain information about its parents or grandparents. Thus, while a lesson typically refers to its activities, it does not contain information as to which curricular unit it belongs. A document's complete lineage, however, can be constructed from all the parent-child relationships stored in the database and hence a listing of 'Related Curriculum' can be extracted from the database, yet not from the document's XML.

Related Curriculum

Subject Areas:	Life Science Science and Technology Biology
Curricular Units:	Floppy Heart Valves Aging Heart Valves
Lessons:	What Do I Need to Know about Heart Valves? Heart to Heart

Figure 4: *TE 1.0 rendering of The Mighty Heart activity's 'Related Curriculum.'*

A second example of database-reliant document rendering in TE 1.0 concerns a document's educational standards. Figure 5 shows the list of K-12 science and engineering standards to which the activity *The Mighty Heart* have been aligned.

Educational Standards

- International Technology and Engineering Educators Association: Technology
  - H. Technological innovation often results when ideas, knowledge, or skills are shared within a technology, among technologies, or across other fields. (Grades 9 - 12) [2000] ...show
- Tennessee: Science
  - Explore the anatomy of the heart and describe the pathway of blood through this organ. (Grades 9 - 12) [2009] ...show
  - Describe the biochemical and physiological nature of heart function. (Grades 9 - 12) [2009] ...show

Figure 5: *TE 1.0 rendering of The Mighty Heart activity's aligned engineering and science educational standards.*

Because the relationship between educational standards and documents is a so-called *many-to-many* one (a standard can

be related to multiple documents and one document can have multiple standards), in TE 1.0 standards were stored uniquely in the database and documents referred to those standards with standard IDs. For *The Mighty Heart* activity the associated XML was as follows[2]:

```
<edu_standards>
    <edu_standard identifier="S11326BD"/>
    <edu_standard identifier="S11326BE"/>
    <edu_standard identifier="S11416DF"/>
</edu_standards>
```

Hence, it is clear that in order to show the information associated with the standard (text, grade level(s), issuing agency, date of issuance, *etc.*), it must be retrieved from the database rather than from the referring document.

TE 2.0 Tagging: Word JSON

While the TE 1.0 tagging process served the TeachEngineering team well, it had a few notable downsides.

-The *Authentic* software to write (tag) the documents in XML needed to be installed on each editor's computer along with the XML schema for each type of curriculum document.

-The editing workflow had a number of steps that required the

2. The S* standard identifiers are maintained by the Achievement Standard Network project. They can be viewed using the following URL: *http://asn.desire2learn.com/resources/S*_code_goes_here*

editor to understand specialized software, including *Authentic* and the *Subversion* version control system.

-Previewing a rendered document required the editor to upload the resulting XML file to the TE site.

-The fact that the spider ran only twice a day limited how quickly new documents (and edits to existing documents) appeared on the site.

One of the goals with TE 2.0 was to streamline the tagging and ingestion process. Since TeachEngineering is a website, the logical choice was to allow editors to add and edit documents from their web browser; no additional software required. As such, TE 2.0 includes a web/browser-based document editing interface that is very similar to that of modern more generalized content management systems such as *WordPress* (Figure 6)

The JavaScript and HTML open source text editor *TinyMCE*, a tool specifically designed to integrate nicely with content management systems, was used as the browser-based editor. TinyMCE provides an interface that is very similar to a typical word processor.

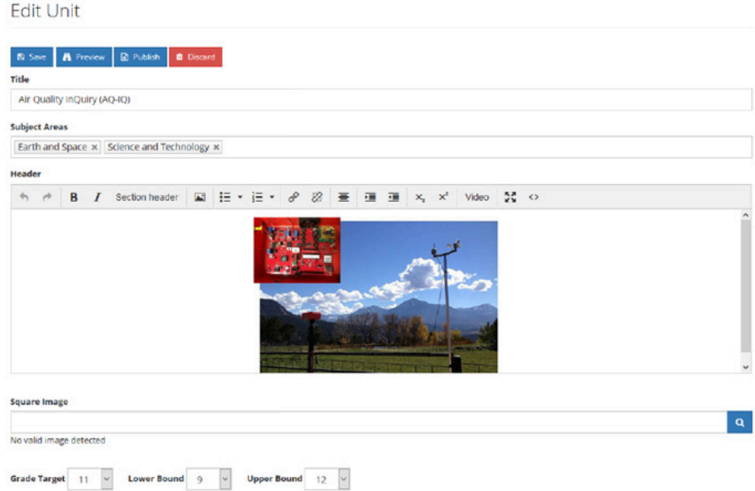


Figure 6: *TE 2.0 document editing interface for a Curricular Unit*

Figure 6 shows an example of editing a document in TE 2.0. The interface provides a few options to support the editor's workflow. The *Save* button saves the in-progress document to the (RavenDB) database. Documents that are in a draft state will not be visible to the public. The *Preview* button shows what the rendered version of the document will look like to end users. The *Publish* button changes the document's status from *draft* to *published*, making it publicly visible. Any errors in the document, such as forgetting a required field are called out by displaying a message and highlighting the offending field with a red border. Publishing of a document with errors is impossible.

As in the case of TE 1.0, documents in TE 2.0 are hierarchically organized in that documents specify their children; *e.g.*, a lesson specifying its activities, or a curricular unit specifying its lessons. But whereas in TE 1.0 editors had to specify these children with a sometimes complex file path, in TE 2.0, they have a simple selection interface for specifying these relationships and are no longer required to know where documents are stored (Figure 7).

Edit Hierarchy

Children of Mars and Jupiter (cub_solar_lesson06)

Curriculum ID to add | + Add Child

Edit | Remove

Edible Rovers (cub_mars_lesson03_activity1)

Rank:

1

Description:

Students act as Mars exploratory rover engineers. Given a budget, parts list and materials constraints, student teams evaluate equipment options and parts, design rovers and build them using cookies, candy, icing, toothpicks and straws. They evaluate their creations at an engineering design review. Easily adaptable for younger students (target grade 7).

Edit | Remove

Are We Alone? (cub_mars_lesson06_activity1)

Rank:

2

Description:

The year is 2032 and your class has successfully achieved a manned mission to Mars! After several explorations of the "red planet," one question is still being debated: "Is there life on Mars?" The class is challenged to conduct a scientific experiment in which they first establish criteria and then evaluate three "Martian" soil samples looking for signs of life. Easily adaptable for younger students (target grade 7).

Figure 7: *TE 2.0's interface for specifying a document's child documents*

One other noteworthy difference between TE 1.0 and TE 2.0's tagging processes is that with TE 1.0, content editors by necessity had to have some knowledge of the internal structure and working of TeachEngineering's architecture. They had to create documents using Authentic, and check the resulting XML document into source control. With TE 2.0, editors edit documents using a familiar WYSIWYG interface. The software behind the scenes takes care of the technical details of serializing the documents to JSON and storing them in RavenDB.

TE 2.0 Document Ingestion and Rendering

With TE 2.0's architecture, the document ingestion and rendering process is greatly simplified. Here we will revisit the ingestion and rendering steps from TE 1.0 and contrast them with the process in TE 2.0.

1. **Document check-in.** In TE 2.0, there is no document check-in process; *i.e.*, no process of moving the file from the local system into the TE repository of documents. When editors save the document they are editing in TE 2.0's web interface, the document is immediately stored in RavenDB.
2. **Metadata generation**. In TE 2.0, there is no separate metadata generation process. As noted in chapter 4, TE 2.0 neither generates nor stores metadata. The JSON representation of the curriculum document is the single version of the TE reality. Whereas TE 1.0 always generated and exposed its metadata for harvesting by meta collections such as the National Science Digital Library (NSDL), TE 2.0 no longer does this. This is mostly because the support for and use of generic metadata harvesting protocols such as OAI-PMH (Open Archive Initiative–Protocol for Metadata Harvesting) have dwindled in popularity.
3. **Document indexing.** There is no document indexing step in TE 2.0. Since documents are immediately saved to RavenDB, there is no need for a separate process to crawl and discover new or modified documents.
4. **Document rendering**. At a high level, the document rendering process in TE 2.0 is quite similar to TE 1.0's process, with a few key differences. For one thing, TE 2.0 was developed in C# as opposed to PHP.
   Whereas in TE 1.0 the hierarchical relationships between any pair of documents were stored as parent-child rows in a relational database table, in TE 2.0, the relationship between all

of the curriculum documents are stored in RavenDB in a single JSON document. This tree-like structure is cached in memory, providing a fast way to find and render a document's relatives (ancestors and descendants). For example, a lesson will typically have one parent curricular unit and one or more child activities. The following is an en excerpt of the JSON document which describes the relationship between documents.


```
{ "CurriculumId": "cla_energyunit",
            "Title": "Energy Systems and Solutions",
            "Rank": null,
            "Description": null,
            "Collection": "CurricularUnit",
            "Children": [
                {
                    "CurriculumId": "cla_lesson1_energyprobl
                    "Title": "The Energy Problem",
                    "Rank": 1,
                    "Description": null,
                    "Collection": "Lesson",
                    "Children": [
                        {
                            "CurriculumId":
                              "cla_activity1_energy_intellig
                            "Title": "Energy Intelligence Ag
                            "Rank": 1,
                            "Description":
                              "A short game in which student
                              find energy facts among a vari
                              of bogus clues.",
                            "Collection": "Activity",
                            "Children": []
                        }
… additional child activities are not shown here for brevity
```


Here you can see that the unit titled *Energy Systems and*

Solutions has a child lesson titled *The Energy Problem*, which itself has a child activity titled *Energy Intelligence Agency*. Since this structure represents the hierarchy explicitly, it is generally a lot faster to extract hierarchical relationships from it than from a table which represents the hierarchy implicitly by means of independent parent-child relationships.

Educational Standards are also handled differently in TE 2.0. As noted earlier in this chapter, curriculum documents in TE 1.0 only stored the identifiers of the standards to which the document was aligned. In TE 2.0, all of the properties necessary to render a standard alignment on a curriculum page are included in the JSON representation of the curriculum document. As discussed in chapter 4, it can sometimes be advantageous to de-normalize data in a database. This is an example of such a case. Since standards do not change once they are published by the standard's creator, we do not need to worry about having to update the details of a standard in every document which is aligned to that standard. In addition, storing the standards with the curriculum document boosts performance by eliminating the need for additional queries to retrieve standard details. Whereas this implies a lot of duplication of standard data in the database, the significant speed gain in extracting the document-standard relationships is well worth the extra storage. The following is an example of the properties of a standard that are embedded in a curriculum document.

```
"EducationalStandards": [
  {
    "Id": "http://asn.jesandco.org/resources/S2454426",
    "StandardsDocumentId":
      "http://asn.jesandco.org/resources/D2454348",
    "AncestorIds": [
      "http://asn.jesandco.org/resources/S2454504",
      "http://asn.jesandco.org/resources/S2454371",
```

```
        "http://asn.jesandco.org/resources/D2454348"
      ],
      "Jurisdiction": "Next Generation Science Standards",
      "Subject": "Science",
      "ListId": null,
      "Description": [
        "Biological Evolution: Unity and Diversity",
        "Students who demonstrate understanding can:",
        "Construct an argument with evidence that in a particu
        habitat some organisms can survive well, some survive
        less well, and some cannot survive at all."
      ],
        "GradeLowerBound": 3,
        "GradeUpperBound": 3,
        "StatementNotation": "3-LS4-3",
        "AlternateStatementNotation": "3-LS4-3"
      }
    ]
```

While the document accessioning experience in TE 2.0 is more streamlined and user friendly, it does have a downside. In TE 1.0, if a property was added to a curriculum document, updating the XML schema was the only step needed to allow editors to utilize the new property. This was because the Authentic tool would recognize the schema change and the editing experience would automatically adjust. In TE 2.0, adding a field requires a developer to make code changes to the edit interface. On balance, however, since document schemas do not change that often, the advantages of a (much) more user-friendly document editing experience outweigh the occasional need for code changes.

7. Why Build Revisited

Why Build… Revisited

Increased Reliance on IS Service Providers

In the *Why Build…* chapter we described how the supply of IS services to buy or rent from IS service providers is increasingly becoming a reasonable alternative for building systems or system components oneself. We mentioned savings in development and maintenance costs and the fact that such external services are typically more reliable and have been much more extensively tested than one's own systems as good reasons to consider buying or renting rather than building.

We also noted that even those who build their own systems rarely build them from the ground up. Just like people who like building their own furniture typically buy their lumber and woodworking tools –of course, some woodworkers grow their own lumber and most woodworkers make at least some tools themselves– system developers typically do not build their own compilers or interpreters, operating systems, database software, web servers and web browsers, *etc.*

Still, when we compare the presence of 'built' rather than 'bought or rented' components in TE 2.0 with those of TE 1.0, we observe a significantly increased reliance on outside service providers. As information systems become more complex in that they provide more functions and as these functions often involve networked and Internet services, system builders increasingly 'outsource' these functions to external service providers. In this (very short) chapter)

we again contrast TE 1.0 with TE 2.0, but this time from the perspective of build *vs.* rent.

TE Searches: From Tool To Service

The chapter on Searching TeachEngineering laid out the various approaches that were used in letting users search the TE collection of documents. Both TE 1.0 and 2.0 used their internally developed codes to search in their databases (tools) for faceted searches; *e.g.*, grade-based searches or standard-based searches. TE 1.0 used a relational database, TE 2.0 a JSON database. However, whereas at first TE 1.0 used a locally installed tool (Lucene) for full text searches, both TE 1.0 and 2.0 ended up relying on external text search services: Google Site Search (now discontinued) in TE 1.0 and Microsoft's Azure Search in TE 2.0. Whereas Google Site Search pricing was based on the number of searches conducted, Azure Search pricing is based on the amount of data stored and indexed.

Educational Standards

Both TE 1.0 and 2.0 relied on the services of D2L's Achievement Standards Network (ASN) for information about K-12 educational (science) standards. ASN's cloud services let users access anything ASN knows about educational standards. Pricing is a fixed annual fee.

Analytics

Both TE 1.0 and 2.0 rely on Google Analytics for analysis of their web

traffic. Whereas TE 1.0 also maintained its own log of all significant website interactions, TE 2.0 no longer stores that information.

Database

As discussed in a previous chapter, rather than using a traditional relational database such as MySQL as we did in TE 1.0, TE 2.0 uses the RavenDB document database. Also in TE 1.0 we chose to install, run, and maintain our own MySQL instance. Doing so, however, implies assuming responsibility for securing, upgrading, and supporting the database which is a significant burden for a small team. In TE 2.0, therefore, we are procuring RavenDB services from RavenHQ, a company affiliated with the developers of RavenDB, to provide us with a fully managed instance of RavenDB. RavenHQ takes care of database updates, security and the server infrastructure required to run RavenDB. Pricing is based on the desired performance level and amount of storage needed and includes the license fee for RavenDB itself.

Social Networking

Whereas TE 1.0 had essentially no social networking aspects, social networking has become an essential and quickly growing component of TE 2.0. The freely available AddThis widget allows users to share TE content via various social media channels such as Twitter, Pinterest, Email, and Facebook.

User Involvement

One very special category of TE users consists of the many authors who have contributed curriculum to TE. As described in the chapter on *Document Accessioning*, in both TE 1.0 and TE 2.0, author submissions are stored and managed in [Open Journal Systems (OJS)](), an open source system for managing peer-reviewed journals. However, whereas under TE 1.0 the TE team ran its own installation of OJS, in the summer of 2017 TE 2.0 switched to using a cloud-based OJS provider. The main reason was that since OJS was and is used purely as a tool for managing curriculum submissions, it was considered cost effective to have a third party run and maintain it. The service follows an annual subscription model with pricing based on the amount of used storage.

Those who use TE as a source of teaching materials comprise, of course, a much larger group. Such users fulfill an important role in that they provide feedback on the functionality of the system, but also on the content of the information served by it. As such, it is in the mutual interest of both users and system maintainers to capture user feedback and to act on that feedback. Of course, some feedback is utter nonsense, disingenuous (refer to *Appendix C: Fake Link Requests*) or consists of nothing but a rant. Much more often, however, user feedback provides an angle on the system's functionality or its content which was missed by its creators or maintainers and which, when taken into account, can lead to functional or content improvements.

TE 1.0 had a simple, internally developed *Contact Us* web page by means of which users could submit any sort of comment or question. These comments were internally processed by the TE team and were not shared on the TE site. Users could also submit reviews of individual lessons or activities through an internally developed *Teacher Reviews* web function. Unlike the *Contact Us* feedback, *Teacher Reviews* were shared on the site.

In TE 2.0 the *Contact Us* facility was retained but other user

involvement functions were greatly expanded. One encompassed switching from the internally hosted *Teacher Reviews* facility to *LinkEngineering*, a cloud-hosted community platform where K-12 educators can share engineering experiences. *LinkEngineering* is a collaborative project of the US National Academy of Engineering and several other engineering-related organizations and is funded by Chevron. As part of an agreement with one of TeachEngineering's funders, we decided to drop our own internal *Teacher Reviews* facility and replace it with the *LinkEngineering* one.

A second change was the deployment of the *Disqus* commenting service, hosted by Disqus.com, on the TE 2.0 curriculum pages. The *Disqus* service collects user feedback in the form of comments and allows other users to search, add and react to those comments with their own, and connects users and their comments with social network platforms. Disqus is free for non-profit organizations.

To further expand our engagement with users, TE 2.0 also uses the email marketing automation platform MailChimp for maintaining a mailing list of users who expressed an interest in receiving our newsletter. MailChimp allows us to create and send email newsletters and other email messages. MailChimp is free for up to 2,000 mailing list subscribers.

Conclusion

Comparing TE 1.0 with TE 2.0, the trend of increasing reliance on outside service providers is clear. Whereas previously we built our own, working but relatively primitive and often buggy services, we nowadays increasingly rely on far more comprehensive and more stable externally provided services. These services not only work better, but they give us more functionality and are typically less expensive than building and maintaining our own.

8. The Develop… Test… Build… Deploy Cycle

Introduction

When developing software, we typically apply some form of a *Develop… Test… Build… Deploy* cycle; a structured progression of work steps from developing code to testing that code, to building a release version of that code, which is then deployed in a production environment. Although these steps are typically executed in succession, steps can be executed multiple times before a cycle completes. For instance, if we find errors in the behavior of our code while testing it, we go back to the *Develop* step to make fixes after which we test again. Sometimes results found at one step may kick us back several steps.

Testing code can happen on several scales and several levels of integration. So-called 'unit tests' test smaller units of code to see if those units behave properly under a variety of circumstances. 'Integration tests' on the other hand test if the larger code complex in which the tested units are integrated works correctly. Of course, this separation of units and complexes is somewhat arbitrary and can be hierarchically layered in good system-theoretical fashion. Whereas several units of code can be integrated into a larger complex, that complex itself can be regarded as a unit in a yet larger complex, in the same way that an element of a system can be a (sub)system itself if we choose to further decompose it.

In order to test the complex, which is composed of units, we must first *Build* it. With that we mean that we must indeed integrate the units into a consistent and (hopefully) working whole. In software engineering the term 'build' is reserved for this integration of codes into such a working whole. This integration step works differently for different types of programming languages, which does not have to concern us for this discussion. For now, it suffices to think of the

build step as the integration of all the units of code, all functions and methods that the system programmers have written, into a cohesive and hopefully working whole.

Regardless of how much we test, however, the likelihood that we have tested all possible code execution paths is small, and hence, the likelihood that some untested code path will at some point in time be executed and cause a problem is larger.[1] Such problems —better known as 'bugs'— may necessitate code fixes and hence, a return to the *Develop* step. However, once code has been put in production —also known as being 'deployed' or 'released' and sometimes as 'shipped'— that code can typically not be taken out of production in order to be repaired. In such cases a parallel develop-test-deploy cycle must be executed and a whole or partial code update must be released.

As we will see in the remainder of this chapter, both TE 1.0 and TE 2.0 applied a structured and carefully followed develop-test-build-deploy process, but TE 2.0's implementation of this process was brought in line with the newer, more modern ways of doing it.

But First: Version Control

Both TE 1.0 and 2.0 rely on a central repository of code shared and agreed upon by all developers. This repository is kept and

1. In computer science, the application of so-called 'formal methods' is aimed at mathematically and logically analyzing and specifying the execution paths of code. This is in contrast to the more common way of simply empirically testing code paths by submitting the code to a variety of specified use cases. Proponents of formal methods propose that the proper application of such methods significantly reduces the likelihood that faulty code execution paths remain undetected prior to code deployment.

maintained in a so-called code-repository or source-control or version-control system. These systems —examples are *Git*, *Subversion*, CVS, *TFS*, *etc*.— manage all changes made to code, allow multiple developers to jointly work on code without overwriting each other's work, and support so-called 'branching;' *i.e.*, the forking of a complex of code into a new complex of code.

In practice it is very difficult to develop and maintain a body of code without using one of these version-control systems. Using them, developers can revert to older versions of code, can track which changes were made by whom and when, can compare different versions of the same code base, line by line and character by character.

These systems also provide protection against multiple developers working on the same code base overwriting each other's work. How can that happen? Easy! Suppose that a certain file contains the code for several methods (functions) and that one developer must work on the code for one of those methods and another developer must work on another method from that same file. It would be quite inefficient if one of these developers would have to wait for the other developer to be done with the file before being able to make code changes. Yet if both developers each work on a copy of the file, there is a very real danger that one merges the modifications back into the code base at time t whereas the other merges his or her code at $t+x$, thereby overwriting the work of the first developer. Version-control systems manage this process by keeping track of who checks in what code at which times. When the system sees a potential cross-developer code override, it flags this as a so-called 'conflict' and gives the developer triggering the conflict several options to resolve the conflict.

Version-control systems also help out a lot with code integration; that 'build' step we mentioned earlier. Suppose, for instance, that a developer writes a new segment of code and that after carefully checking and testing it (s)he checks the code into the version-control system. Between the time the developer started working on this code and the time (s)he checks it in, other developers made

changes to existing code and added code of their own. Hence, it is possible that the new code checked in last 'breaks the build;' *i.e.*, that it is not functionally compatible with the rest of the code. Version-control systems provide at least one means of avoiding and one means of mitigating this problem. Developers can update their local copy of the code they have not worked on and test their additions to see if they cause any problems (avoidance). On the integration/build side of the process we can revert to a previous version of the code which caused the build to brake and report to the developer who broke the build that (s)he must modify the new code so that it no longer breaks the build (mitigation).

TE 1.0 used the CVS version-control system early on, but migrated to *Subversion* a few years later. TE 2.0 uses *Git*.

TE 1.0 Develop-Test-Deploy

The TE 1.0 develop-test-build-deploy process was effective and simple, although perhaps not maximally flexible. The process consisted of three steps:

- Step 1. Sandbox or development site coding and testing. New code, code adjustments and code extensions are developed on an internal system. For TE 1.0 this was a web site which, although visible and accessible to the world, was anonymous in that no links on the web pointed to it. Such an internal system or site is typically called a sandbox (developers are free to 'play' in it). In TE 1.0 we called our sandbox our 'new' site.
  Depending on how a software development group sets up its sandboxing, individual developers can have their own, individual sandbox, or, as in the TE 1.0 case, they can share a common sandbox. Obviously, individual sandboxes provide more opportunities to work on code without impacting other developers. The TE 1.0 team, however, was small enough that a single common sandbox, in combination with a version-control system, worked just fine.Code developed in the sandbox would typically be reviewed by TE 1.0 project members for functional adequacy and robustness. Once approved, the code would be

moved to the next step, namely the 'test' site.

- Step 2. Integration testing. Beside the (shared) sandbox, TE 1.0 maintained a release test site. This site —software, database content and document repository— was synchronized with the production/release site, but was used for testing all new and modified code against the complete system. Hence, once sandbox code was approved for release, it was deployed on the test site for integration testing. The time it would take to conduct this integration testing varied from just a few minutes for a simple user interface change; *g.*, a color change or fixing a typo, to a day or longer for testing new or updated periodic back-end processes.
  Only once the software was verified to operate correctly on the 'test' site, would it be released in production. In case errors were found, the process returned to the sandbox stage.
- Step 3. Production/Deployment. Releasing sandbox-approved and test site-verified code was quite easy because it consisted of deploying the test site-verified code from the updated version-control system to the production site.

***All is Flux, Nothing Stays…* TE 1.0 Continuous System Monitoring**

One good way to experience Heraclitus's famous "*All is Flux, Nothing Stays*" or to experience the universal phenomenon of system entropy, is to release code into production and sit back and wait for it to stop functioning. Although the stepped process of Sandbox à Test site à Production site is relatively safe in that it limits the risk of releasing faulty or dysfunctional code, it is always possible, and indeed likely, that at a later stage —sometimes months later— a problem emerges. This can happen for a variety of reasons. Perhaps a developer relied on a specific file system layout which later on became invalid. Or perhaps code relies on pulling data from an external service which for some reason or other suddenly stops or seems to stop working (Did we not pay our annual license fee? Did we run out of our free allocation of search queries? Is

the service still running? Did the service change its API without us making the necessary adjustments?).

Experienced software developers have great appreciation and awareness of the principles of permanent flux and system entropy and hence, will make sure that they build and deploy facilities which continuously monitor the functioning of their systems. Of course, these monitoring facilities need some monitoring themselves as well. Although at least in theory this leads to an infinite regress, monitoring the monitoring processes can mostly be accomplished through simple and often manual procedures which can be integrated into a team member's job responsibilities. Table ?? contains a list of TE 1.0's (automated) system monitoring processes.

Monitor	*Frequency*	*Details*
Systems up test	Once per minute	A simple test to see if our Website and database are up and running.
Regression tests	Every 12 hours	Tests for most new features and all bug fixes are run in sequence. The following is the summary of the last TE 1.0 regression test run on April 28, 2016: *Start Time: Thu Apr 28 04:00:04 2016* *Total Run Time: 357.458 seconds* *Test Cases Run: 139* *Test Cases Passed: 137* *Test Cases Failed: 2* *Verifications Passed: 268* *Verifications Failed: 2* *Average Response Time: 2.544 seconds* *Max Response Time: 85.099 seconds* *Min Response Time: 0.002 seconds*
Link diagnostics	Every 12 hours	Test all Web links on the TE pages and report failing links (the link, the source of the link, the contributor of the source, the error code associated with the failed link)
HTML diagnostics	Once a month	Run an HTML checker on a random sample of Web (static and dynamic) web pages.
Metadata harvesting checker	Once a month	A process which queries sites which harvest our content, making sure that the sites continue to harvest our content.

TE 2.0 Develop-Test-Deploy

The development, testing, and deployment process in TE 2.0 is similar to that of TE 1.0. The differences are in the details.

- Step 1. *Development*. In TE 2.0 developers code new features on their local PC using a shared (development) instance of the database. For larger teams, it would be better for developers to have their own development copy of the database to allow work to be done in isolation. However, due to the very small size of the TE 2.0 development team, a shared development database has not been problematic.New features are developed as branches off of the main Git code repository branch. This allows new features to be developed in isolation from the production code base until they are ready to be released.A key aspect of developing new features is the development of corresponding unit tests, code that tests that a particular unit of code behaves as intended. As of November 2017, the TE 2.0 code base comprises 395 unit tests for server-side C# code and 313 unit tests for client-side JavaScript code. Beyond verifying that code behaves as intended today, unit tests also make it safer to make changes to code in the future. Without unit tests, it is very difficult to ensure that code changes do not break existing functionality.
- Step 2. *Integration*. When new feature development has progressed to the point where it is ready to be included in the next production release, it is merged onto the master branch of the code repository. That is, the changes from the feature branch are applied to the master branch. This triggers an automated process which compiles the code, runs the unit tests, and, if all of the unit tests pass, deploys the code to a development instance of the site. The code is compiled in 'Debug' mode which includes debugging information and un-minified[2] JavaScript code to assist with debugging and troubleshooting. As features that are to be included in the next release are merged onto the master branch, this integration

testing ensures that all of the code changes play well together.

- Step 3. *Beta testing*. Once integration testing verifies that the code compiles and the automated tests pass, the master branch is merged into the QA (quality assurance) branch. This triggers another automated build and deployment process. This time, the code is built in 'Release' mode which results in compiled code that does not have debugging information embedded and JavaScript which has been minified and obfuscated[3][3]. The output of the build is deployed to a beta instance of the TeachEngineering site. This is a place where the entire TE team can review and test changes to the site. This process is also known as acceptance testing.
- Step 4. *Staging*. Once a set of code changes is ready to be released, it is merged from the QA branch onto a branch called *Release*. This triggers another build process that results in the code being deployed to a staging environment which is an exact duplicate of the production environment and uses the production database. Additionally, this build process ends with

2. Minification is the process of removing all unnecessary characters from source code; e.g., spaces, new lines and comments. In the case of JavaScript (which runs in the browser), this speeds up the transfer of the code from the server to the client without affecting its functionality. Minified code, however, is difficult to read for humans, hence us using un-minified code for debugging purposes.
3. Code obfuscation refers to the practice of purposefully rendering source code difficult to read for humans, typically in order to make it more difficult for ill-willed individuals to search for weaknesses and security exploits.

a series of 'Smoke Tests' which perform automated browser-based testing using Selenium, a tool for web browser scripting. These tests exercise key functionality of the site to ensure that nothing is broken; *e.*, on fire.

- Step 5. *Release*. Once the staging site is verified to be working correctly, it is swapped with the production site. That is, all production traffic is redirected to the staging site, which becomes the new production site. In the event a problem is encountered after release that necessitates a roll back, it is easy to redirect traffic back to the prior version of the site.

In TE 2.0, each of these steps is entirely automated and can be initiated by executing one or two command line statements. Automation is key to having quick, repeatable, and error-free releases. This automation allows updates to TE 2.0 to be released frequently, as often as once a day or more. Releasing software updates more frequently results in smaller, less-risky updates. Frequently integrating and releasing code is known as 'Continuous Integration' and 'Continuous Deployment.' Prior to the widespread adoption of these two practices, integration and releases would happen much less frequently, often as infrequently as once a quarter. This resulted in increased risk and longer feedback cycles.

Behind the Magic Curtain

There is a lot going on for each of the develop-test-deploy steps described in the previous section. Code is retrieved from source control, compiled, tested, and deployed. The process is highly automated, and thus can seem somewhat magical at first glance. Not too long ago, setting up an automated build and deployment process like this required setting up, configuring, and maintaining a build server such as Jenkins or TeamCity as well as a server for running the chosen source control system. Similarly, hosting development, testing, and production instances of an application would typically involve buying, configuring, and maintaining multiple servers.

With the emergence of software-as-a-service and the 'cloud,' it

is no longer necessary to configure and maintain the basic infrastructure; *i.e.*, hardware and software needed to develop, deploy, and host applications. For example, TE 2.0 utilizes Visual Studio Team Services (VSTS) to host its Git repository and perform the build and deployment process. As a cloud-hosted solution, VSTS saves the TE team from having to maintain source control and continuous integration servers.

Similarly, the development, beta, staging, and production environments are hosted in Azure, Microsoft's cloud hosting platform. TE 2.0 uses Azure's Platform as a Service (PaaS) offering known as Azure App Service. With PaaS, the cloud provider takes care of maintaining and updating the server and operating system that runs the application. In effect, everything below the application layer is abstracted away and managed by the cloud hosting provider. This is especially beneficial for a small team such as the TE team. Instead of worrying about operating system updates and hardware maintenance, our limited resources can be focused on activities that make TE a better product.

Azure App Service also provides a number of other value-added capabilities. For example, if there is a sudden surge of traffic to the TE site, Azure App Service will automatically add more server capacity. When traffic levels subsequently drop to a level that does not require additional capacity, the extra capacity is withdrawn. Server capacity is billed by the minute and you only pay for capacity when you are using it. This is one of the key benefits of hosting applications in the cloud. In a traditional hosting model, one would have to pay up front for the server capacity needed for peak load, even if it is unused a vast majority of the time. With Platform as a Service, capacity can be added and removed as demand warrants.

The ability to deploy code to a staging site and swap it with the production site as described in steps 4 and 5 of the previous section is also a feature of Azure App Service. This can be done with just a few clicks or with a single command-line statement.

TE 2.0 Continuous System Monitoring

Azure App Service also provides a number of capabilities for

monitoring application health. For TE 2.0, the site is configured to send an alert via e-mail if certain adverse events happen. These include heavy CPU load, heavy memory usage, excessive 500 errors, and slow site response.

In addition, TE 2.0 uses Azure Application Insights, an Application Performance Management tool. Application Insights captures detailed data about application performance, errors, and user activity. This data is fed into a web-based dashboard. It also uses machine learning to detect events such as slow performance for users in specific geographic locations or a rise in the number of times a specific error happens. Application Insights has also been configured to access TE from five different geographic locations every five minutes. An email alert is generated if 3 or more of the locations are unable to access the site.

TE Meta Monitoring

Besides pure functional aspects of system performance, there are other, higher level (or 'meta') aspects which need regular reporting and checking. In today's web- and internet-based world, one of these aspects is whether third parties which drive traffic to our system; *i.e.*, search engines such as Google, know about your content and assess our content as an attractive target. This information obviously must come from the parties owning the search engines; it is not information internal to our system. Fortunately, search engines such as Google often make their diagnostics tools available so that as content providers we can know how the search engines assess our content. In Google's case one of those tools is *Google Search Console*[4]. This tool provides lots of information on how Google harvests your content. Needless to say, then, that periodic monitoring of Google Search Console, either manually or by using its API, provides valuable information on how well your site is viewed by the world's most popular search engine.

4. At the time of TE 1.0, this service was known as *Google Webmaster Tools.*

Another, valuable meta monitoring service is [*Google Analytics* (GA)](). GA is a service to which one can report requests coming into one's website. GA keeps a record of those requests and reports them back on demand using any number of user-chosen facets. For example, one may ask for a timeline of requests, for any time frame and pretty much any time step. One can also ask for a breakdown by technology, browser, operating system, location, page, *etc*. Obviously, a lot of useful information will be hiding in these data. Both TE 1.0 and TE 2.0 use(d) GA.

A third type of meta monitoring simply collects and reports aggregate information about a system. For TeachEngineering this means being able to tell how many items the library has at any moment for any grade or combination of grades in the K-12 grade range or how many different institutions have contributed curriculum and how much they have contributed.

Appendix A: When Editing Code Files, Use a Text Editor; Not(!) a Word Processor

On several occasions in this text you will encounter exercises which require you to create files containing some textual content; perhaps a PHP program, some XML, some JSON, *etc*. Although there are several ways to create a file with text in it, most of us will use a software application in which we type or copy/paste the text. When you do this in the context of this book's exercises, we want you to use a so-called *text editor*, not a *word processor*.

For a quick lookup of common free and open source text editors for Windows, macOS and Linux, see the table at the end of this appendix. If you want to understand the difference between *text editors* and *word processors*, read on.

A *Word Processor* and a *Text Editor* Are Not the Same Thing

From experience we know that few students who are new to coding realize that what looks to be text to them, does not look the same to a machine. For instance, suppose that someone types the following into the interface of a text editing program such as Windows' Notepad: *foo Enter goo*

No-one will be surprised when it looks like the following on the screen:

```
foo
goo
```

Let us assume that we now store these two lines of text in a file called *foo.txt*.

When asked how many symbols (letters, bytes) *foo.txt* contains, most people will say "six;" three for the word *foo* and three for the word *goo*. However, if one would actually check how many bytes the file *foo.txt* contains, one might find one of several answers, none of which is "six."

Let's see what happens when on a Linux machine we create the file (not using Notepad, because that is not a Linux text editor) and we ask how many bytes the resultant file has. One of the Linux commands for doing this is *wc* (for *w*ord *c*ount):

```
>wc foo.txt
2 2 8 foo.txt
```

wc tells us that the file has two lines (the first "2"), two words (the second "2") and eight ("8") characters.

Another way is to check the byte count with the *ls* command:

```
>ls -l foo.txt
-rw-rw-r--. 1 userid userid 8 Nov 20 14:22 foo.txt
```

Notice the '8' indicating the file size in bytes.[1]

To see each of the bytes in the file, we use the *od* (octal dump) command:

```
>od -c foo.txt
f   o   o  \n   g   o   o  \n
```

1. In these examples, characters are represented with single bytes; *i.e.*, one byte per character. However, in a character representation system such as UTF-16, characters are represented by two bytes each. Hence, this would double the counts used here.

Sure enough, the file has eight one-byte characters in it. Six to form the words *foo* and *goo* and two so-called new-line characters (*n*). The effect of these new-lines, of course, is that when you pull up the file in a text editor, both *foo* and *goo* are on their own lines.

If we try this on Windows, however, we get a different result (we once again read the files with the *od* command in Linux AFTER we have created the file in *Notepad*):

```
>od -c foo.txt
f   o   o  \r  \n   g   o   o
```

We once again have eight bytes, but this time *foo* and *goo* are separated by a return character (*r*) and a new-line character (*n*), while there is no new-line after *goo*.

If you find this confusing, look at the size of the file after we type the exact *foo* and *goo* text in *Ms Word* and store it in a file *foo.docx*:

```
>ls -l foo.doxc
-rwxr-xr-x. 1 userid userid 11561 Nov 20 14:46 foo.docx
```

Holy, moly!! This time we end up with 11,561 bytes!

So what is going on here? Programs such as Microsoft's *Word* and Apple's *TextEdit* are *word processing* programs. Their job is two-fold: store text and format that text in any way the user specifies. This formatting can take lots of forms, from changing font type and font size, to line indentations, inter-line spacing, *etc*. Since all this formatting information must be stored with the text, word-processed texts typically have far more many bytes that text alone. Hence the 11,000+ bytes for the simple foo/goo *Word* file.

If, on the other hand, we use a *text editor* (not a word processor), such as *Notepad*, *Notepad++* , *TextMate*, *Sublime*, *nano*, *emacs*, *vi*, *bluefish* or one of a host of other ones, all we get out is plain,

unformatted text[2]. If in those texts we want some indentation at the beginning of a line, we must type spaces and tabs, and if we want a new line, we must type the newline character (*Enter* key) and those are stored in the file, but nothing else.

Whereas this distinction between word processors and text editors explains the size and content differences between the files generated by them, it does not yet explain why the foo/goo text files generated with a text editor in Linux and Windows are different (same size, different content). That difference is explained by Linux and Windows following different conventions for storing plain text files.

Which Text Editor to Use?

In this text, you will only(!) use a text editor. Coders who write program code, write their code using text editors, not word processors. As explained above, word processors are used to make text look good for humans to read. Program code, however, needs no formatting other than distributing it over multiple lines and adding some indentation; all of which can be easily accomplished with the enter, space, and tab keys[3].

This does, however, not mean that coders are impartial about

2. Files with unformatted text are known as 'plain text' files.
3. Many text editors can be set to automatically adjust text layout associated with a specific programming language; *e.g.*, C, C++, HTML, Python, *etc*. In these cases, the text editor automatically includes spaces, new-lines, tabs and even parentheses and curly braces, each of which will, of course, become part of the text.

which text editor to use. Most coders are very attached to and enamored with their favorite text editor and will stick to it unless something much better appears.

We, as authors of this text have no axe to grind as to what text editor you use. However, we want you to not(!) use a word processor, as this will likely cause all different sorts of problems when you try to run the exercises. Unfortunately, recognizing whether or not you have a word processor or a text editor is not always easy, especially in confusing cases such as Apple's *TextEdit*, which is not(!) a text editor but a word processor, so better not use it because it will not allow you to store plain text files.

The following table should help you find a few suitable (free and open source) text editors. Once again, as authors we really do not care which one you use. Just try one. If you like it, stick with it. If you do not like it, grab another one (Just do not spend all your time comparing specs between these, only to find out at the end that you have no time left to actually use them).

Table 1: (very) limited list of free and open source text editors

	notepad++	*bluefish*	*TextMate*	*vi*	*emacs*	*nano*
Windows	x	x		x	x	
MacOS			x	x	x	
Linux		x		x	x	x

Appendix B: (Unintended?) Denial of Service Attack

The Attack

On Jan. 26, 2015, a little before 3 pm, some of the student workers working on the TE 1.0 system informed us that the system had slowed down to a trickle. A few minutes later, the systems group hosting the TE (1.0) Web site informed us that both the HTTP request rate and the associated request wait times as well as database query times had all greatly increased. For individual users accessing the system, request wait times had grown so much that, as far as they were concerned, the system had halted.

Although symptoms such as these can have a variety of causes, one likely candidate is a so-called *Denial of Service (DoS) attack*. In a DoS attack, requests for service arrive at a machine at a rate which is higher than the rate at which the requests can be served. In process management language: the arrival rate outstrips the service rate. In any capacity-constrained system (we see the same when the rate of customers arriving at a restaurant outstrips the rate at which these customers are served, or when the rate of cars arriving at a highway on-ramp outstrips the capacity of the highway to absorb these vehicles and move them along) this results in longer wait times or more precisely, queueing times. Since the requests are waiting in the queue to be served, the issuer of the request has the impression that the entire service is halted, just like the driver of a car stopped in a traffic jam on a busy road has the impression that the road is blocked and vehicles 'requesting' passage are not at all served. When this principle is abused and lots of requests are purposefully directed at a service in order to overwhelm its service capacity, we call it a *Denial of Service (DoS) attack*. If these requests

are arriving form a large number of different machines, we speak of a *Distributed DoS* (*DDoS*).

As our Jan. 26th, 2015 DoS incident illustrates, however, not all DoS instances originate as targeted attacks.

Although our TeachEngineering server served a variety of protocols and hence, a possible DoS could target any of these services, we took a look at its Apache Web server log to see if it would contain information about what was going on. The following is a random one-second excerpt from those logs:

```
113.248.198.22 - - [26/Jan/2015:10:00:40 +0000]
"GET /announce.php?info_hash=
%A8%82c%F92%7F9%150%A9%112%10%CF%0C%0E%D8d%87s&peer_id=
%2DSD0100%2D%EC%9D%CB%BD%A7%2D%E5%EBw%A2F%BD&ip=
113.248.198.22&port=13337&uploaded=1005870892&downloaded=
1005870892&left=706329&numwant=200&key=2497&compact=
1&event=started HTTP/1.0" 302 587 "-" "Bittorrent"

117.79.232.6 - - [26/Jan/2015:10:00:40 +0000]
"GET /announce.php?info_hash=
%B8%A7%7B%11%9D%F2m%E8%EE%92%A8%DA%2Dxy%11%94%F8Z%E9&peer_id=
%2DUT3000%2D0%1C%D5%23%3A%92%5B%B0%BC%2ExO&ip=
192.168.1.104&port=1080&uploaded=0&downloaded=0&left=
289742100&numwant=200&key=644621065&compact=1&event=
started HTTP/1.0" 302 578 "-" "Bittorrent"

222.78.27.77 - - [26/Jan/2015:10:00:40 +0000]
"GET /announce.php?info_hash=
%8F%A6%81%3A%B7%2C%C1%C8%D1v%25%F8%B75Z%D2I%84%07H&peer_id=
%2DSD0100%2D%C3%26v%06%94%DB%29%CA%DD%84%C7%7B&ip=
222.78.27.77&port=19678&uploaded=125304832&downloaded=
125304832&left=878468806&numwant=200&key=31199&compact=
1 HTTP/1.0" 302 575 "-" "Bittorrent"

111.4.119.139 - - [26/Jan/2015:10:00:40 +0000]
```

```
"GET /announce.php?info_hash=
%DC%C9%A9%2Cwl%ED%7F%0Fmm%21p%D1%01%0C7%16%EFk&peer_id=
%2DSD0100%2D%9CkT%08%92%F2%CC%A8%AC%9E%00%7B&ip=
192.168.21.52&port=8123&uploaded=23068672&downloaded=
23068672&left=814367656&numwant=200&key=19228&compact=
1 HTTP/1.0" 302 566 "-" "Bittorrent"

115.201.102.64 - - [26/Jan/2015:10:00:40 +0000]
"GET /announce?info_hash=
%B5%F6%9CL%BF%A4%D0%2D%08%D7%13%070k%9C%80%29M%BA%BA&peer_id=
%2DSD0100%2D%21%B3Q%22O%BF%28%22%B5%8F%40q&ip=
115.201.102.64&port=13318&uploaded=1062366471&downloaded=
1062366471&left=1040028409&numwant=200&key=5854&compact=
1 HTTP/1.0" 302 572 "-" "Bittorrent"

222.131.158.50 - - [26/Jan/2015:10:00:40 +0000]
"GET /announce?info_hash=
z6%D5%97g%C1%9CIS%9B%10%F4%C0%8B%C1%99%AF%09g%C3&peer_id=
%2DSD0100%2D%D1%D4%E9%CF%1Ay%86%C7z%8F%95%F1&ip=
222.131.158.50&port=11338&uploaded=7219559940&downloaded=
7219559940&left=15781004769&numwant=200&key=17705&compact=
1 HTTP/1.0" 302 573 "-" "Bittorrent"

116.21.125.252 - - [26/Jan/2015:10:00:40 +0000]
"GET /announce?info_hash=
%A1%D0%BDy%FA%D7%27u%0A%96%8D%FDSb%EB%BF%8C%F3%EC%AD&peer_id=
%2DSD0100%2D%1AS%8356%28%06%AEe%05%E7%E6&ip=
192.168.1.99&port=13897&uploaded=149680706&downloaded=
149680706&left=72221773&numwant=200&key=30556&compact=
1 HTTP/1.0" 302 563 "-" "Bittorrent"
```

The log entries are easy to parse:

```
IP-address_of_the_requestor – - [date and time of the request]
"HTTP_request_method
/requested_file?URL_parameters HTTP_version" HTTP_response_code
```

```
number_of_bytes_returned "-" "User_agent"
```

Looking at this series of requests, things started to become clear. The machines making the requests were all based in China (you can geographically trace these IP's at www.yougetsignal.com/tools/visual-tracert) and the requests all came from a software identifying itself as *BitTorrent*, a well-known file-sharing protocol.

From this we concluded that somehow –most likely by accident but possibly on purpose– our TeachEngineering machine had become registered to be part of the BitTorrent file-sharing network and we were being flooded with BitTorrent requests.

Now what?

Once we concluded that the problem was caused by a flood of BitTorrent requests coming from China, we had to decide on a remedy. The easiest and most obvious course of action would have been to block all BitTorrent requests. This would probably have worked just fine, but since we did not know whether the inclusion of our IP in the BitTorrent network was accidental or purposeful, we erred on the side of caution. We guessed that if the attack was purposeful, blocking the requests might anger (or challenge) the perpetrator(s), as a result of which we might become the target of more vicious attacks. Hence, rather than blocking the requests we decided to 'deflect' them. We replied to the requests with a page without content but which resulted in a 200 (Success) HTTP response code. Since serving these 'null' pages required very little effort from our server and raised the service rate considerably, this approach solved the problem for our users.

An alternative –and in hindsight perhaps preferable– strategy would have been to have the server reply with an HTTP 404 or 410 error. A 404 error signals the requester that the requested

file cannot be found whereas a 410 indicates that the requested resource is no longer available.

What most likely happened

We have learned since that our TeachEngineering machine was very likely involved in an incident where Turkish and Chinese DNS servers performed a kind of DNS spoofing, a process which replaces server IP addresses with other, non-related ones. Here is the text from a Jan. 2015 jwz.org blogpost by Jamie Zawinsky.

> "*After a bit of logging and searching I found out that some Chinese ISP (probably* CERNET *according to the results of whatsmydns.net) and some Turkish ISP (probably* TTNET*) respond to dns queries such as a.tracker.thepiratebay.org with various IPs that have nothing to do with piratebay or torrents. In other words they seem to do some kind of* DNS Cache Poisoning *for some bizarre reason.*
>
> *So hundreds (if not thousands) of bitTorrent clients on those countries make tons of 'announces' to my web servers which result pretty much in a* DDoS *attack filling up all* Apache*'s connections.*
>
> So *basically, entire countries' worth of porn hounds randomly start hammering on my server all at once, even though no* BitTorrent *traffic has ever passed to or from the network it's on, because for some unknown reason, the now-long-defunct piratebay tracker sometimes resolves to my* IP *address. Hooray.*"

TE 2.0 and DoS?

Although there is no reason to expect that TE 2.0 is less likely to be targeted by another DoS attack, either deliberately or by mistake, it does seem reasonable to expect that it being hosted in Microsoft's Azure cloud should provide it with better, more robust protection than the 1.0 version which was hosted at the university. Without suggesting that the protection provided by a university lab or service is inherently insufficient, it stands to reason that large cloud service providers expend a lot of effort on keeping their renters safe from the vagaries of today's Internet. Hence, it might be a good idea to host services such as TE in an environment which puts a premium on safety.

Appendix C: Fake Link Requests

My name is Kelly: or How to Prey on People's Vanity and Love of Children and Good Causes[1]

On October 8, 2013 the following email arrived at the TeachEngineering project:

> *From: kellyh@enrichingkids.com*
> *[mailto:kellyh@enrichingkids.com]*
> *Sent: Tuesday, October 08, 2013 2:38 PM*
> *To: TeachEngineering.org*
> *Subject: feedback and a thank you for your green info*
>
> *Good Evening –*
>
> *My name is Kelly. I work with kids in a youth activities program in Montpelier, Vermont. Recently a lot of questions*

1. The cases described here were recorded over several years. Although they are true recordings, recreating or following them might not be possible anymore since fraudulent individuals and endeavors frequently change their identity and hidings. Also, some of the domain names associated with these attempts have changed ownership since we recorded these attempts.

have come up about ways we can help the environment and decided to do a project on eco-friendly transportation. At the end of our project we're going to compile everyones research into a packet to be distributed to earth science classes this fall. This morning I came across your page xxx and wanted to thank you. It has some good information ways to reduce your impact on the environment that we're going to include in our packet.

My junior counselor Julianne, also found a site that has some great information on eco-friendly transportation and alternative fuels (automotivetouchup.com/touch-up-paint/green-is-more-than-a-paint-color-for-cars); would you mind adding it to your page if it's not too much trouble? It has some great resources not listed on your site and I'd like to show Julianne and her professor that her hard work is paying off.

Let me know what you think and if you get a chance to update. Enjoy your week.:)

-Kelly

On first inspection, this email seems perfectly legitimate. It is polite, friendly, flattering for sure, and it sounds quite convincing. Moreover, the return email address looks bonafide, the reference to a TeachEngineering lesson is perfectly integrated into the text and the link requested to be included in TeachEngineering looks innocuous and to-the-point. Indeed, this must surely be one of the more clever attempts at infiltrating a website. One of us even admits that he fell for it until a colleague pointed out that we had encountered a similar attempt at infiltration before.

So what is the problem here? Very little, on the face of it. But what if the request to include the link is merely an attempt to sneak an advertisement onto our pages? Or worse: what if the content

to which the link points now, is changed to something a lot more nefarious once the link is included on our pages?

Inspecting the link automotivetouchup.com/touch-up-paint/green-is-more-than-a-paint-color-for-cars (at least at the time of this writing) shows a page with seemingly innocent materials and text referring to electrical and hybrid cars. However, it also contains links to paint products and links to pages which themselves contain links to products; *i.e.*, materials visible to a spider/crawler.

So, we decided to look just a little deeper and try to figure out who or what 'Kelly' is. Kelly's email comes from *enrichingkids.com*, so we pulled that up in our browser.

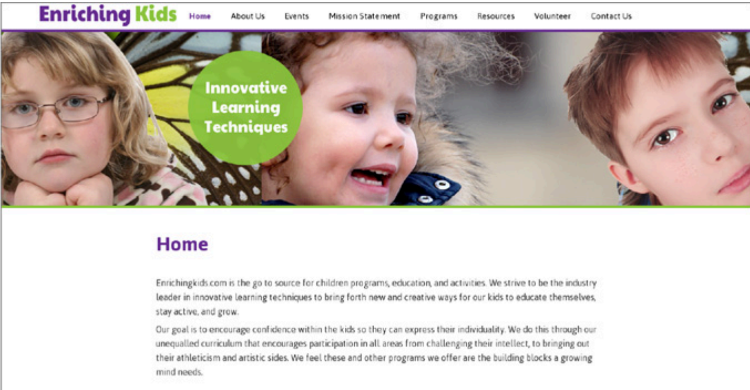


Figure 1: *Home page of enrichingkids.com.*

Again, this looks innocent enough. But then, when we start pulling up some of the 'tab' pages, we note that although the data and information on the tabs is innocent, they contain very little if any information: no mission statement, no terms-of-use policy, no ownership or organizational information. Nothing, really. Just placeholder sentences such as "*We strive to provide lifelong learning opportunities and are pleased to have the opportunity to work with you and your children.*" True, the 'Resources' page points to about 30 seemingly legitimate resources, but these are all elsewhere on the

web and not owned by enrichingkids.com. Of course, most of the links on Google's search pages also point to materials not owned by Google, but then again, Google does not ask us to be linked.

So it seems that enrichingkids is either a legitimate, though rather clumsy web site, or it might be a dummy site meant to convince those who do not immediately trust that Kelly indeed works hard for kids.

But who or what is enrichingkids.com really? Who, for instance owns the domain name? A quick 'whois' search yields the following:

Domain Name: *enrichingkids.com*

Creation Date: 23 Jun 2000 19:08:00

Registrant Name: WHOIS AGENT

Registrant Organization: WHOIS PRIVACY PROTECTION SERVICE, INC.

Registrant Street: PMB 368, 14150 NE 20TH ST – F1

Registrant Street: C/O ENRICHINGKIDS.COM

Registrant City: BELLEVUE

Registrant State/Province: WA

Admin Email: HDDQCHRX@WHOISPRIVACYPROTECT.COM

This is where the plot thickens. *HDDQCRX* is a strange email name, so who or what is WHOISPRIVACYPROTECT.COM (WHOIS PRIVACY PROTECTION SERVICE, INC)? Pulling it up in the browser gives the answer: "*Whois Privacy Protect offers a premium service to domain name registrants to protect their personal information from being displayed in the public Whois database.*"

Here again, we have no proof that Kelly is not at all interested in kids and merely fishing for ad exposure (or worse). After all, domain name owners may have good reasons to hide their identity from the public. But by now we have encountered just a few too many items of non-information. 'Kelly' has no last name and does not identify herself in the email. The "*youth activities program in Montpelier, Vermont*" is not identified. Neither are Julianne or her professor. The enrichingkids.com site has no real information and has all the characters of a dummy or ghost site. And the domain name's owner is hiding him/her/itself behind a whois-masking service.

Clever, for sure. Actually, quite a bit cleverer than the Nigerian offering you millions in exchange for a small donation. Just not clever enough, though …and plenty sleazy.

Unfortunately, these fraudulent attempts at link intrusion, possibly because of their cleverness and resultant success rate, are increasing in frequency. Below are two more examples. You might want to track down the origin of the emails and see what you find there! Pay close attention to linguistic clues in the messages. Whereas the spelling errors (typos) may lend a sense of authenticity to the emails, the grammar errors are a clear sign that something is amiss. Similarly, nonsensical references such as Sheryl's "*simple machines field trip*" reveal their fraudulent nature.

From: heather.graham@cmufsd.org

Sent: Wednesday, March 21, 2012 8:12 AM
To: TeachEngineering.org
Subject: Suggestions and Compliments on your site, www.teachengineering.org!

Good morning & Happy Spring!

My name is Heather and I teach at Cleary Mountain Elementary School in Virginia. I wanted to take a few minutes to write to you because my students and I found your webpage xxx very helpful! We have been using your resources as a reference for our Recycling project in class!

My student, Erika, has been using another page that was very helpful that she brought to my attention:

"Environmental Concerns – Recycling"

http://www.aaenvironment.com/environmental-concerns-recycling.htm

I was wondering if you would mind adding it to your page?

We both thought it would be a perfect addition to your collection of resources and I know that Erika would be delighted to see her suggestion up on your page!

I have also decided that Erika will be receiving bonus points on her next test for her newly discovered resource so thanks so much for contributing to her eduction! ? We look forward to hearing from you and thank you again!

Heather

From: Sheryl Wright [mailto:swright@goodwincc.org]

Sent: Wednesday, February 13, 2013 7:40 AM

To: TeachEngineering.org

Subject: a quick thanks for your helpful simple machine resources… ?

Hi,

I just wanted to take the time to contact you and let you know that my classmates and I have really enjoyed using your page xxx for our simple machines field trip and projects. My teacher, Mrs. Wright, thought it would be nice if we wrote you a thank you note (using her email) to let you know that it's been such a great help for us ?

As a small token of our appreciation, we all thought it would be nice send along another resource that we came across during our project: http://www.directfitautoparts.com/simple-machines-used-in-autos.html It has some helpful information and sites to

learn all about simple machines (wheels, axles, levers, pulleys, etc.) that we thought could help other students as well.

And if you decided to add it to you other resources, I'd love to show Mrs. Wright that the site was up to share with other students learning about simple machines ?

But thanks again for your help! And I hope to hear back from you soon.

Sincerely,

Emma Hanley (and the rest of Mrs. Wright's class)

***From:** teachengineering-request@lists.colorado.edu*

***On Behalf Of** teachengineering (noreply)*
Sent: Tuesday, August 16, 2016 11:25 AM
To: teachengineering@lists.colorado.edu
Subject: Contact Us Feedback

Name: Morgan Konarski
Email: m.konarski@safekidsusa.net
Comments:

Hi there, I just wanted to send you a quick email on behalf of my son Christian. Christian is currently participating in "Camp Grandma" while my husband and I are at work. While at "Camp Grandma" my mother tries and finds fun things for the two of them to do during the day that are both fun and educational. This week, my mother decided to teach Christian all about STEM and the opportunities kids have in all of those subjects. Christian has said he's always wanted to be an engineer, so my mother has been trying to find fun games

and resources for them to check out. Christian was so excited and fascinated that he insisted on doing research of his on last night on engineering and careers in this industry. He came across your page https://www.teachengineering.org/k12engineering/what and told me how helpful and easy to understand your page was. As a mother, I just wanted to thank you for making it and your help in encouraging my son with your resources. He also came across this great article with a lot of info about STEM careers, engineering basics, and full of STEM resources. Christian thought it might be a great addition to the links and resources on your page. Here it is if you wanna check it out "Computer and STEM Careers for Kids" https://www.vodien.com/blog/education/computer-stem-careers.php Would you consider adding it for me? I would love to surprise him and show him that his research will help other kids learn all about STEM in a fun way! Thank you so much, Morgan Konarski

Appendix D: I am robot...

A Word on *robots.txt*

The *robots.txt* file, when inserted into the root of the web server's file system, can be used by web designers and administrators to communicate instructions to robot software on how to behave. For instance, web administrators may wish to exclude certain pages from being visited by robots or may want to instruct robots to wait a certain amount of time before making any subsequent requests.

Instructions in the robot.txt file follow the so-called *robot exclusion protocol* or *robot exclusion standard* developed by Martijn Koster in 1994. Although the protocol is not part of an official standard or RFC, it is widely used on the web.

Why do we care to provide instructions to robots visiting our web pages? Well, we generally welcome those robots which, by extracting information from our pages, may give back to us in the form of high rankings in search engines; *e.g.*, Googlebot and Bing. Yet, we often want to prevent even these welcome robots from visiting certain sections of our web site or specific types of information. For instance, we might have a set of web pages which are exclusively for administrative use and we do not want them to be advertised to the rest of the world. It may be fine to have these pages exposed to the world for some time without enforcing authentication protocols, but we just do not want them advertised on Google. Similarly, we might, at one time or other, decide that an existing set of pages should no longer be indexed by search engines. It might take us some time, however, to take the pages down. Instructing robots in the meantime not to access the pages might just do the trick.

Naturally, the rules and instructions coded in a *robots.txt* file cannot be enforced and thus, compliance is entirely left to the robot

and hence, its programmer. 'Good' robots always first request the *robots.txt* file and then follow the rules and directives coded in the file. Most (but not all) 'bad' or 'ill-behaved' robots –the ones which do not behave according to the rules of the *robots.txt* file– do not even bother to request the *robots.txt* file. Hence, a quick scan of your web server log to see which robots requested the file is a good (although not perfect) indication of the set of 'good' robots issuing requests to your site.

The two most important directives typically used in a *robots.txt* file are the *User-Agent* and *Disallow* directives. The *User-Agent* directive is used to specify the robots to which the other directive(s) are directed. For instance, the directive

```
User-Agent: *
```

indicates 'all robots,' whereas the directive

```
User-Agent: googlebot
```

specifies that the directives apply to *googlebot*.

Different directives can be specified for different robots. For instance, the content:

```
User-agent: googlebot

Disallow: /private/

User-agent: bing

Disallow: /
```

disallows *googlebot* the *private* directory whereas *bing* is disallowed the whole site.

Googlebot is a pretty 'good' robot, although it ignores certain types of directives. It ignores, for instance, the *Crawl-delay* directive which can be used to specify time (in seconds) between requests, but at least it says so explicitly and publicly in its documentation.

Sniffing Out Robots

Robots also can be a drain on data communications, data retrieval and data storage bandwidth. About 2/3 of all HTTP requests TeachEngineering receives originate from robots. Hence, we might benefit (some) from limiting the 'good' robots from accessing data we do not want them to access, thereby limiting their use of bandwidth. Of course, for the 'bad' robots the robots.txt method does not work and we must resort to other means such as blocking or diverting them rather than requesting them not to hit us.

Regardless of whether or not we want robots making requests to our servers, we might have good reasons for at least wanting to know if robots are part of the traffic we serve and if so, who and what they are and what percentage of the traffic they represent:

- We might want to try and separate humans —you know, the creatures which Wikipedia defines as *"...a branch of the taxonomical tribe Hominini belonging to the family of great apes"*— from robots, simply because we want to know how human users use our site.
- We might want to know what sort of burden; *e.g.*, demand on bandwidth, robots put on our systems.
- We might just be interested to see what these robots are doing on our site.
- *Etc.*

We can try separating robots from humans either in real time; *i.e.*, as the requests come in, or after the fact. If our goal is to block robots or perhaps divert them to a place where they do not burden our systems, we must do it in real-time. Oftentimes, however, separation can wait until some time in the future; for instance when generating periodic reports of system use. In many cases a combination of these two approaches is used. For instance, we can use periodic after-the-fact web server log analysis to discover the

robots which visited us in a previous period and use those data to block or divert these robots real-time in the future.

Real-time Robot Sniffing:

Several methods for real-time robot sniffing exist:

- **JavaScript-based filtering**. Robots typically (although not always!) do not run/execute the JavaScript included in the web pages they retrieve. JavaScript embedded in web pages is meant to be executed by/on the client upon retrieval of the web page. Whereas standard web browsers try executing these JavaScript codes, very few robots do. This is mostly due to the fact that the typical robot is not after proper functionality of your web pages, but instead is after easy-to-extract information such as the content of specific HTML or XML tags such as the ones containing web links. Hence, we can make use of snippets of JavaScript code which will most likely only be executed by browsers driven by humans. This is, for instance, one reason why the hit counts collected by standard Google Analytics accounts contain very few if any robot hits. After all, to register a hit on your web page with Google Analytics, you include a snippet of JavaScript in your web page containing the registration request. This snippet, as we just mentioned, is in JavaScript and hence, is typically not executed by robots. There is, of course, no guarantee. For instance, we can write a robot program which makes HTTP requests at web sites through a regular browser. In that case, the browser just acts as a front to the robot, but the JavaScript will get executed and the robot makes it through the filter.
- **Real-time interrogation**. Sometimes we just want to be very confident that a human is on the other end of the line. This is the case, for instance, when we ask for opinions, customer

feedback or sign-up or confidential information. In such cases, we might opt to use a procedure which in real-time separates the humans from the machines, for instance by as asking the requester to solve a puzzle such as arranging certain items in a certain way or recognizing a particular pattern. Of course, as machines get smarter at solving these puzzles, we have to reformulate the puzzles. A good example is Google's moving away some time ago from a puzzle which asked requesters to recognize residential house numbers on blurry pictures. Whereas for a while this was a reliable human/machine distinguishing test, image-processing algorithms have become so good that this test is no longer sufficient.

- **Honeypotting**. Another way of recognizing robots in real-time is to set a trap into which a human would not likely step. For instance, we can include an invisible link in a web page which would not likely be followed by a human but which a robot which follows all the links on a page —a so-called crawler or spider— would blindly follow.
  We successfully used this technique in TE 1.0 where in each web page we included an invisible link to a server-side program which, when triggered, would enter the IP address of the requester in a database table of 'suspected' IPs. Every month, we would conduct an after-the-fact analysis to determine the list of (new) robots which had visited us that month. The ones on the 'suspect' list were likely candidates. Once again, however, there is no guarantee since nothing prevents an interested individual to pull up the source of an HTML page, notice the honeypot link and make a request there, just out of curiosity.
- **Checking a blacklist**. It may also be possible to check if the IP of an incoming request is included in one or more blacklists. Consulting such lists as requests come in may help sniffing out returning robots. Blacklists can be built up internally. For instance, one of the products of periodic in-house after-the-fact robot sniffing is a list of caught-in-the-act robots.

Similarly, one can check IPs against blacklisted ones at public sites such as http://www.projecthoneypot.org/
Of course, checking against blacklists takes time and especially if this checking must be done remotely and/or against public services, this might not be a feasible real-time option. It is, however, an excellent option for after-the-fact robot sniffing.

After-the-fact Robot Sniffing

Just as for real-time robot sniffing, several after-the-fact —*ex post* if you want to impress your friends— robot detection methods exist.

- **Checking a blacklist.** As mentioned at the end of the previous section, checking against blacklists is useful. If the IP address has been blacklisted as a robot or bad robot, it most likely is a robot.
- **Statistics**. Since robots tend to behave different from humans, we can mine our system logs for telltale patterns. Two of the most telling variables are hit rates and inter-arrival times; *i.e.*, the time elapsed between any two consecutive visits by the same IP. Robots typically have higher hit rates and more regular inter-arrival times than humans. Hence, there is a good chance that at the top of a list sorted by hit count in descending order, we find robots. Similarly, if we compute the variability; *e.g.*, the standard deviation of inter-arrival times and we sort that list in ascending order —smallest standard deviations at the top— we once again find the robots at the top. Another way of saying this is that since robots tend to show very regular or periodic behavior, we should expect to see very regular usage patterns.

Thought Exercise D.1:

Here we explore this statistical approach for a single month; *i.e.*, April 2016 of TE 1.0 usage data. Assume a table in a (MySQL) relational database called *april_2016_hits*. Assume furthermore that from that table we have removed all 'internal' hits; *i.e.*, all hits coming from inside the TeachEngineering organization. The *april_2016_hits table* has the following structure:

Field	Type	Nullability	Key
id	int(11)	no	primary
host_ip	varchar(20)	no	
host_name	varchar(256)	yes	
host_referer	varchar(256)	yes	
file	varchar(256)	no	
querystring	varchar(256)	yes	
username	varchar(256)	yes	
timestamp	datetime	no	

- Find the total number of hits:

  ```
  select count(*)
  from april_2016_hits;
  ```

  777,018

- Find the number of different IPs:

  ```
  select count(distinct(host_ip))
  from april_2016_hits;
  ```

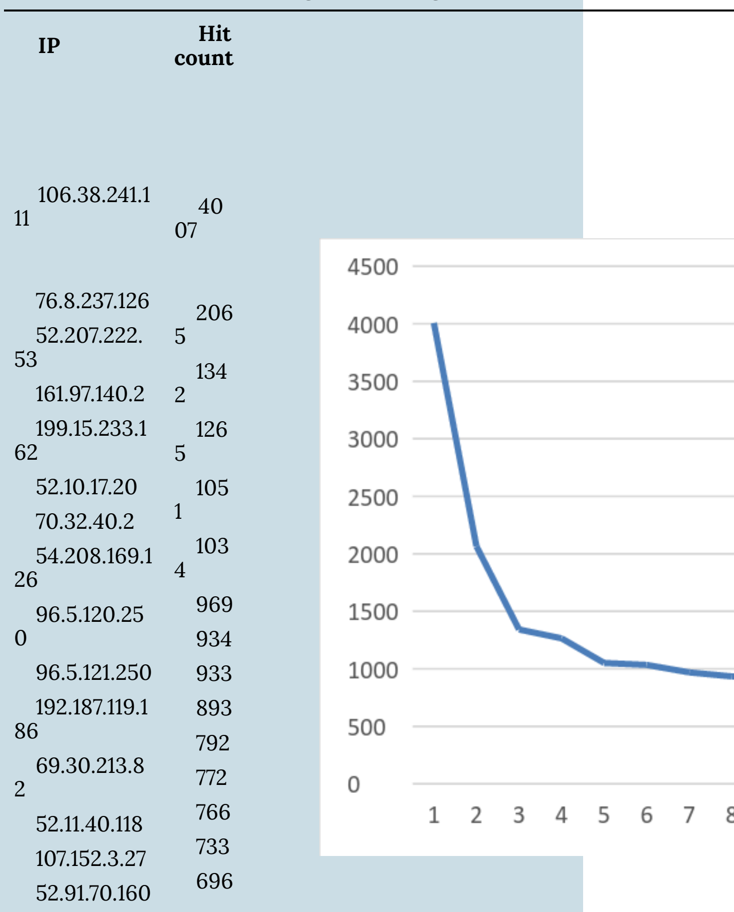
  296,738

- Find the 15 largest hitters and their hit counts in descending order of hit count:

```
select host_ip, count(*) as mycount
from april_2016_hits
group by host_ip
order by mycount desc
limit 15;
```

Figure 1: 15 largest April 2016 hitters and th

IP	Hit count
106.38.241.111	4007
76.8.237.126	2065
52.207.222.53	1342
161.97.140.2	1265
199.15.233.162	1051
52.10.17.20	1034
70.32.40.2	969
54.208.169.126	934
96.5.120.250	933
96.5.121.250	893
192.187.119.186	792
69.30.213.82	772
52.11.40.118	766
107.152.3.27	733
52.91.70.160	696



Notice the exponential pattern (Figure 1).

- Let us first see if we can track these top two

IPs down a bit (https://www.ultratools.com/tools/ipWhoisLookupResult):

- 106.38.241.111: CHINANET-BJ, Beijing, China. projecthoneypot.org flags this IP as a likely robot (possible harmless spider)
- 76.8.237.126: TELEPAK-NETWORKS1, C-Spire Fiber, Ridgeland, MS. projecthoneypot.org does not have data on this IP.

◦ Next, let us take a look at the hit patterns. The first 100 timestamps by 106.38.241.111 in April 2016:

```
  select timestamp
from april_2016_hits
where host_ip = '106.38.241.111'
order by timestamp desc
limit 100; --(The inter-arrival times were computed
   from the data returned
from the database query)
```

Day	Time	Inter-arrival time	Inter-arrival time (secs)
4/8/2016	11:29:47		
4/8/2016	11:31:53	0:02:06	126
4/8/2016	11:33:04	0:01:11	71
4/8/2016	11:36:24	0:03:20	200
4/8/2016	11:37:33	0:01:09	69
4/8/2016	11:38:42	0:01:09	69
4/8/2016	11:39:50	0:01:08	68
4/8/2016	11:41:00	0:01:10	70
4/8/2016	11:42:09	0:01:09	69
4/8/2016	11:43:20	0:01:11	71
4/8/2016	11:45:44	0:02:24	144
4/8/2016	11:46:52	0:01:08	68
4/8/2016	11:48:02	0:01:10	70
4/8/2016	11:49:12	0:01:10	70
4/8/2016	11:50:25	0:01:13	73
4/8/2016	11:51:33	0:01:08	68
4/8/2016	11:52:43	0:01:10	70
4/8/2016	11:53:51	0:01:08	68
4/8/2016	11:55:02	0:01:11	71
4/8/2016	11:57:46	0:02:44	164
4/8/2016	11:58:54	0:01:08	68
4/8/2016	12:00:09	0:01:15	75
4/8/2016	12:01:24	0:01:15	75
4/8/2016	12:02:36	0:01:12	72
4/8/2016	12:03:48	0:01:12	72

4/8/2016	12:05:00	0:01:12	72
4/8/2016	12:07:19	0:02:19	139
4/8/2016	12:08:32	0:01:13	73
4/8/2016	12:09:42	0:01:10	70
4/8/2016	12:10:52	0:01:10	70
4/8/2016	12:12:04	0:01:12	72
4/8/2016	12:13:18	0:01:14	74
4/8/2016	12:15:40	0:02:22	142
4/8/2016	12:16:47	0:01:07	67
4/8/2016	12:17:55	0:01:08	68
4/8/2016	12:19:03	0:01:08	68
4/8/2016	12:20:12	0:01:09	69
4/8/2016	12:21:22	0:01:10	70
4/8/2016	12:22:38	0:01:16	76
4/8/2016	12:23:46	0:01:08	68
4/8/2016	12:24:59	0:01:13	73
4/8/2016	12:27:14	0:02:15	135
4/8/2016	12:28:23	0:01:09	69
4/8/2016	12:33:00	0:04:37	277
4/8/2016	12:34:08	0:01:08	68
4/8/2016	12:37:29	0:03:21	201
4/8/2016	12:38:41	0:01:12	72
4/8/2016	12:39:48	0:01:07	67
4/8/2016	12:40:56	0:01:08	68
4/8/2016	12:42:05	0:01:09	69

When studying the inter-arrival times; *i.e.*, the time periods between hits, we notice that at

least in the first 100 hits, there are no hits within the same minute. When checked over the total of 4,007 hits coming from this IP, we find only 46 hits (1.1%) which occur within the same minute and we find no more than two such hits in any minute. Such a regular pattern is quite unlikely to be generated by humans using the TeachEngineering web site. Moreover, the standard deviation of the inter-arrival times (over the first 100 hits) is 38.12 seconds, indicating a very periodic hit frequency. When we remove the intervals of 100 seconds or more, the standard deviation reduces to a mere 2.66 seconds and the mean inter-arrival time becomes 70.46 seconds.

Finally, Figure 2 shows a histogram of the inter-arrival times for the first 100 hits. It indicates regular clustering at multiples of 70 minutes (70, 140, 210 and 280 minutes).

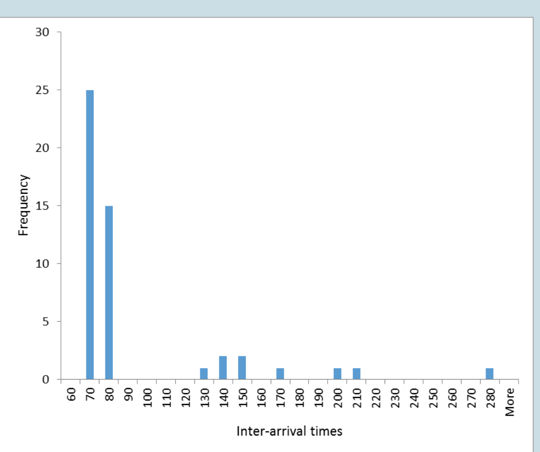


Figure 2: *histogram of the inter-arrival times for the first 100 hits by IP 106.38.241.111.*

These data indicate a very periodic behavior which makes intra-minute hits very rare indeed. From these data, plus the fact that this IP has been flagged by projecthoneypot.org, we can conclude that this IP represented a robot.

- For IP 76.8.237.126 we find the following distribution of hits over the days in April:

```
select day(timestamp), count(*) from april_2016_h
where host_ip = '76.8.237.126'
group by day(timestamp)
order by day(timestamp);
```

Day (date)	Hit count
1	1
4	9
5	1436
6	577
7	2
8	1
11	10
12	2
13	8
14	1
15	6
19	10
22	2

We notice that 97.5% (1436+577)/2065) of this IP's activity occurred on just two days: April 5th and 6th.

Looking at the 1,436 hits for April 5th, we find a mean inter-arrival time of 18.39 seconds and an inter-arrival time standard deviation of 194 seconds. This is odd. Why the high standard deviation? Looking through the records in the spreadsheet (not provided here) we find five pauses of 1000 seconds or more. After eliminating these, the mean drops to 8.14 seconds and the standard deviation reduces to a mere 16 seconds.

These data are compatible with at least two scenarios:

1. A robot/spider/crawler which either pauses a few times or which gets stuck several times after which it gets restarted.
2. A two-day workshop where (human) participants connect to TeachEngineering through a router or proxy server which, to the outside world, makes all traffic appear as coming from a single machine.

Looking a little deeper, the 1000+ second pauses mentioned above appeared at typical workday break intervals: 8:00-9:00 AM, 12:00-1:00 PM, *etc*.

At this point we looked at the actual requests coming from this IP to see if these reveal some sort of pattern we can diagnose. We retrieved a list of requests, over both days in April, ordered by frequency:

```
select file, count(*) from april_2016_hits
where host_ip = '76.8.237.126'
and (day(timestamp) = 5 or day(timestamp) = 6)
group by file
order by count(*) desc;
```

The results are telling:

File	Hit count
/livinglabs/earthquakes/socal.php	1034
/index.php	476
/livinglabs/earthquakes/index.php	395
/livinglabs/index.php	30
/view_activity.php	29
/livinglabs/earthquakes/sanfran.php	8
/livinglabs/earthquakes/japan.php	6
/livinglabs/earthquakes/mexico.php	6
/browse_subjectareas.php	4
/browse_lessons.php	4
/googlesearch_results_adv.php	4
/login.php	3
/whatisengr.php	2
/whyk12engr.php	2
/history.php	2
/ngss.php	2
/browse_curricularunits.php	2
/search_standards.php	1
/about.php	1
/view_lesson.php	1
/googlesearch_results.php	1

The vast majority of requests are associated with a very specific and small set of

TeachEngineering activities, namely the *Earthquakes* Living Lab, with more than 40% of the hits requesting the home pages of TeachEngineering (*index.php*) and the Earthquakes lab (*/livinglabs/earthquakes/index.php*). Whereas this is not your typical robot behavior, it is quite compatible with a two-day workshop on earthquake-related content where people come back to the system several times through the home page, link through to the Earthquakes Lab and take things from there.

Creative Commons License

This work is licensed by René Reitsma under a Creative Commons Attribution-NonCommerical-Share Alike 4.0 International License (CC BY-NC-SA)

You are free to:

Share – copy and redistribute the material in any medium or format

Adapt – remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial – You may not use the material for commercial purposes.

ShareAlike – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions – You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Recommended Citations

APA outline:

Source from website:

- (Full last name, first initial of first name). (Date of publication). Title of source. Retrieved from https://www.someaddress.com/full/url/

Source from print:

- (Full last name, first initial of first name). (Date of publication). Title of source. Title of container (larger whole that the source is in, i.e. a chapter in a book), volume number, page numbers.

Examples

If retrieving from a webpage:

- *Berndt, T. J. (2002). Friendship quality and social development.* Retrieved from insert link.

If retrieving from a book:

- Berndt, T. J. (2002). Friendship quality and social development. *Current Directions in Psychological Science*, 11, 7-10.

MLA outline:

Author (last, first name). Title of source. Title of container (larger

whole that the source is in, i.e. a chapter in a book), Other contributors, Version, Number, Publisher, Publication Date, Location (page numbers).

Examples

- Bagchi, Alaknanda. "Conflicting Nationalisms: The Voice of the Subaltern in Mahasweta Devi's Bashai Tudu." *Tulsa Studies in Women's Literature*, vol. 15, no. 1, 1996, pp. 41-50.
- Said, Edward W. *Culture and Imperialism.* Knopf, 1994.

Chicago outline:
Source from website:

- Lastname, Firstname. "Title of Web Page." Name of Website. Publishing organization, publication or revision date if available. Access date if no other date is available. URL .

Source from print:

- Last name, First name. *Title of Book*. Place of publication: Publisher, Year of publication.

Examples

- Davidson, Donald, *Essays on Actions and Events*. Oxford: Clarendon, 2001. https://bibliotecamathom.files.wordpress.com/2012/10/essays-on-actions-and-events.pdf.
- Kerouac, Jack. *The Dharma Bums*. New York: Viking Press, 1958.

Versioning

This page provides a record of changes made to this guide. Each set of edits is acknowledged with a 0.01 increase in the version number. The exported files for this toolkit reflect the most recent version.

If you find an error in this text, please fill out the form at bit.ly/33cz3Q1

Version	Date	Change Made	Location in text
0.1	**MM/DD/YYYY**		