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 lots of 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 and Design textbooks.

Textbook	Pages relational	Pages NoSQL
Valacich, J.S., George, J.F., Hofer, J.A. (2015) *Essentials of Systems Analysis and Design.* Pearson.	Chapter 9: 45 pp.	0 pp
Satzinger, J., Jackson, R., Burd, S. (2016) *Systems Analysis and Design in a Changing World.* CENGAGE Learning.	Chapter 9 : 35 pp.	0 pp.
Tilley, S., Rosenblatt, H. (2017) *Systems Analysis and Design.* CENGAGE Learning.	Chapter 9: 45 pp.	0 pp.
Dennis, A., Wixom, B.H., Roth, R.A. (2014) *System Analysis and Desig*n. 6th ed. John Wiley & Sons.	Chapter 6 & 11: 40 pp.	1 pp.
Dennis, A., Wixom, B.H., Tegarden, D. (2012) *System Analysis & Design with UML Version 2.0.* John Wiley & Sons.	Chapter 9: 40 pp.	2 pp.
Dennis, A., Wixom, B.H., Tegarden, D. (2015) *System Analysis & Design. An Object-Oriented Approach with UML. 5th ed.* John Wiley & Sons.	Chapter 9: 22 pp	2 pp.
Coronel C., Morris, S. (2016) *Database Systems. Design, Implementation and Management.* Cengage learning.	700+	11 pp.

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.

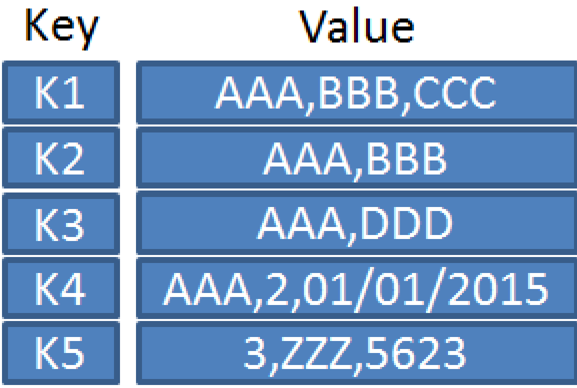


Figure 1: *Lookup structure of a key-value store (source: https://en.wikipedia.org/wiki/File:KeyValue.PNG)*

Of the six texts listed in Table 1, only Dennis, Wixom & Tegarden (2012, 2015) and Coronel & Morris (2016) 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 also 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 document formats and then kept their metadata for sorting and searching, many modern NoSQL document databases 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 plain 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 much) 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 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

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

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 lightweight; *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 exists 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. One example of those is Yahoo Query 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.
- In this list we have not mentioned issues associated with data volume and complexity scaling and distributed computing; *i.e.*, having to or benefiting from breaking up a 'database' in several or many logically united but spatially separated components. Much of the claimed benefits of NoSQL over relational databases address those issues. Although these fascinating and important issues certainly belong in database textbooks and course materials,, they are outside of the scope of our TE-specific discussion.

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-

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.

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 supports 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 resource structure and to manage resource content validity (refer to Chapter 3 for details on XML and validity checking). Consequently, we needed a way to index the content of the resources 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 resources was still quite fluid. Although we had settled on the main resource 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 would have to be added in the first few years of operations. Moreover, we considered it quite likely that in the future entirely new docuresource 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 resource data, yet the resources' 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:Remove (comment out) the `foreach()` loop and replace it with the following:

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

- The first step consists of a traditional, hardwired schema of meta (master) tables which are populated

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

with instructions on how the actual resource database must be constructed.
- In a second step, a program extracts the instructions from the meta tables, builds the data tables accordingly and then processes the resources, 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 resource 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.

TE 1.0 Data Tables

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

- Although the various resource 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 resource-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 resource 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 resources and K-12 educational standards is a many-to-many one, a resource-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 resources; for instance, a curricular unit resource 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 resources may contain links to other TeachEngineering resources 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 resource-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)Remove (comment out) the `foreach()` loop and replace it with the following: `Console.WriteLine(resultList.Count);`	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**	**nullable**	**datatype**
19	child_document	/child_documents/link	group	NULL	NULL
24	cost_amount	/activity_cost/@amount	number	yes	float
25	cost_unit	/activity_cost/@unit	string	yes	varchar(100)
6	edu_std	/edu_standards/edu_standard	group	NULL	NULL
1	edu_std_id	/@identifier	string	yes	varchar(8)
33	engineering_category	/engineering_category_TYPE/@category	string	yes	varchar(250)
14	grade_lowerbound	/grade/@lowerbound	number	yes	int(10)
13	grade_target	/grade/@target	number	no	int(10)
15	grade_upperbound	/grade/@upperbound	number	yes	int(10)
26	keywords	/keywords/keyword	group	NULL	NULL
27	keyword		string	yes	Varchar(250)

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 resource* has a *link_text*, a *link_type* and a *link_url*. Note that this information is similar to that contained in the resources' 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 resource. For example, the cost associated with an activity (*cost_amount*) is a *float*, can be null (*yes*) and can be extracted from the XML resource with the XPath expression */activity_cost/@amount*. Similarly, the *grade_target* of a resource is an *int*(10), cannot be null (*no*) and can be extracted with the XPath expression */grade/@target*. Note that array-like data types 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 resource-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 resources, 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* resource 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 resource 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 resource 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 *sprinkle* table with the requisite columns, their declared data types 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 resources' structure such as the addition of the *sprinkle* resource type. During those years the collection grew from a few hundred to over 1,500 resources. Yet when the decision was made to rebuild the system, the architectural choice of having a separate store of XML-based resources to be indexed into a relational database was questioned in light of the newly available NoSQL document databases. Why not, instead of having two more or less independent representations of the resources (XML and relational database), have just one, namely the resource database; *i.e.*, a database which

houses the resources themselves. If that database of resources can be flexibly and efficiently searched, it would eliminate a lot of backend software needed to keep the resource repository and the database in sync with each other. Better still, when rendering resources 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 resource 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, resources 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 resource, the resource'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 resources in the database; *e.g.*, title, target grade, required time, summary, keywords, *etc.*, but also the entire text of resources. 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 resource content from resource searching since the basic data of these databases are the resources themselves. Hence, we have everything we want to know about these resources 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!

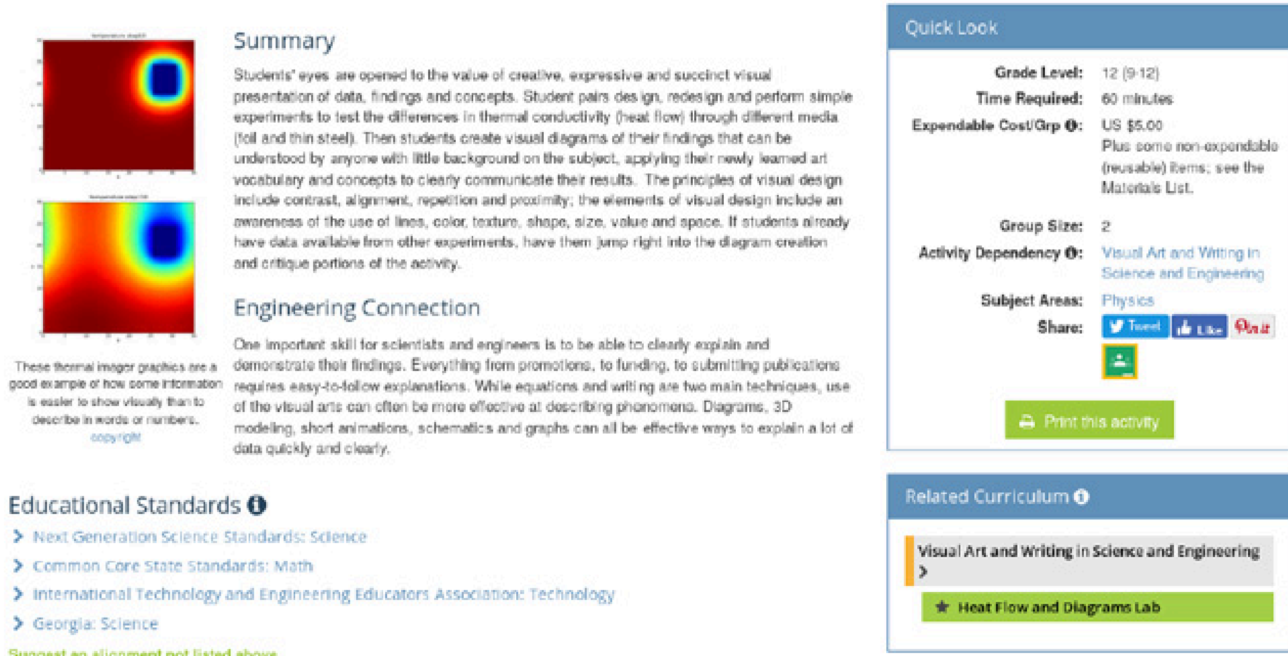


Figure 2: *Heat flow activity lists Related Curriculum*

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 resource 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 resource is made available it would be perfectly acceptable if that resource is not immediately available everywhere in the world. Moreover, due to the relatively small number of resources 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 a JSON Document Store: MongoDB

In the remainder of this chapter we work through a practice examples using the well-known JSON document store *MongoDB*. 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' resources. Sprinkles are abbreviated versions of TeachEngineering activities.

Exercise 5.1: Setting Up MongoDB

We will install the MongoDB 5.x (Community) Server locally. MongoDB also offers free-of-charge use of its Atlas cloud offering, but Atlas set up is more involved that installing MongoDB on a local machine which takes just a simple download and install.

We assume a Windows machine, but installs for other platforms are available as well.

- Download the *MongoDB Community Server* version (as *msi* file) from https://www.mongodb.com/try/download/community (Note: these installation instructions might be slightly different for newer versions of MongoDB).
- Run the downloaded installation script:
    - Accept the license.
    - Select the *Complete* (not the *Custom*) version.
    - Select *Run service as a Network Service user.*
    - Uncheck the *Install MongoDB Compass* option.
    - Select *Install.*
    - *Finish* the install.
- Find the folder 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* runs the server instance
- *mongo.exe* runs a Mongo command line interpreter (CLI) through which we can type and send commands to *mongod*.

- Use the *Windows Task Manager* to see if the process *MongoDB Database Server* (*mongod.exe*) is running. If not, start *mongod.exe*.
- 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 CLI and run the command:

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

We will communicate with MongoDB in two modes: first, by sending it commands directly using its CLI and then programmatically using Python.

Exercise 5.2: MongoDB Command Line Interpreter (CLI)

We will run some MongoDB commands through MongoDB's CLI. We will run without access control; *i.e.*, without the need for a username and password.[5]

- Run *mongo.exe* to start a MongoDB CLI. Enter all commands mentioned below in this CLI (hit *return/ enter* after typing the command).

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

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

```
use foo
```

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

```
db.createCollection("Swatches")
```

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

Add four documents to *Swatches* using the following four commands:

5. Note: running a database server without login and password protection is obviously not good practice. However, for our case it makes things much simpler. Just do not forget to shut down or even uninstall your Mongo server when you're done.

```
db.Swatches.insertOne( { "swatch": "cotton_swatch",
 "qty": 100, "tags": ["cotton"],
 "size": { "h": 28, "w": 35.5, "uom": "cm" } })
db.Swatches.insertOne( { "swatch": "wool_swatch",
 "qty": 200, "tags": ["wool"],
 "size": { "h": 28, "w": 35.5, "uom": "cm" } })
db.Swatches.insertOne( { "swatch": "linen_swatch",
 "qty": 300, "tags": ["linen"],
 "size": { "h": 28, "w": 35.5, "uom": "cm" } })
db.Swatches.insertOne( { "swatch": "cotton_swatch",
 "qty": 100, "tags": ["cotton"],
 "size": { "h": 50, "w": 50, "uom": "cm" } })
```

Now, let us see what we have in *Swatches* using the *find()* command. Since we pass no criteria to *find()*, all documents in *Swatches* are returned:

```
db.Swatches.find()
```

Which *swatches* are made of *cotton*?

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

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

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

Note how in each of these cases we provide the *find()* command with a pattern to match.

To close the CLI, either close the CLI window or type Cntrl-C.

Exercise 5.3: Python (3.*) — 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 Python (3.*) as our language, but MongoDB can be programmatically accessed with other languages as well.

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

```
[
```

```
{ "swatch": "nylon_swatch", "qty": 100,
  "tags": ["nylon"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } },
{ "swatch": "rayon_swatch", "qty": 200,
  "tags": ["rayon"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } },
{ "swatch": "suede_swatch", "qty": 300,
  "tags": ["suede"],
  "size": { "h": 28, "w": 35.5, "uom": "cm" } },
{ "swatch": "nylon_swatch", "qty": 100,
  "tags": ["nylon"],
  "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 https://jsonformatter.org/json-parser.

In order for our program to interact with MongoDB, make sure that you have the Python *pymongo* package installed.

Since in the previous exercise (*Exercise* 5.2) we already made a database (*foo*) with a *Swatches* collection, we will reuse that collection and programmatically add four swatches to it.

Here is the Python (3.*) program for inserting the four documents stored in the *swatches.json* file into the `Swatches` collection:

```
 import json
import pymongo

#Load the JSON data from the file
#Assuming c:\temp\swatches.json -- Change if needed
json_file = "c:\\temp\\swatches.json"
try:
  in_file = open(json_file, "r")
except Exception as err:
  print("Error opening JSON file...", err)
  exit(1)
my_json = in_file.read()
in_file.close()

#Load JSON object from the my_json string
json_data = json.loads(my_json)
```

```
#Connect to MongoDB without(!!) user credentials.
try:
  my_client = pymongo.MongoClient("localhost")
except Exception as err:
  print("Error connecting to MongoDB...", err)
  exit(1)

#Declare the database
my_db = my_client["foo"]

#Declare the collection
my_collection = my_db["Swatches"]

#Loop through the json_data list and store each
# element (swatch) into my_collection.
for swatch in json_data:
  try:
    my_collection.insert_one(swatch)
  except Exception as err:
    print("Error inserting swatch...", err)
    exit(1)
```

Run the program.

If you still have your MongoDB CLI open (if not, just start it again — see previous section on how to do this), let us see what the program accomplished:

```
use foo

db.Swatches.find()
```

Next, we run a program which queries the `Swatches` collection for swatches of which we have more than 100 (*qty* > 100). Here is the program:

```
import json
import pymongo

#Connect to MongoDB without(!!) user credentials.
try:
  my_client = pymongo.MongoClient("localhost")
except Exception as err:
  print("Error connecting to MongoDB...", err)
```

```
  exit(1)

#Declare the database
mydb = my_client["foo"]

#Declare the collection
mycollection = my_db["Swatches"]

#Query for swarches with qty > 100
try:
  my_swatches = mycollection.find({"qty" : {"$gt" : 100 }})
except Exception as err:
  print("Error querying swatches...", err)
  exit(1)
for swatch in my_swatches:
  print(swatch)
```

Exercise 5.4: Python (3.*) — MongoDB Programmatic Interaction: TeachEngineering 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 resources. Our sample contains six TE 2.0 *sprinkle* resources and is stored at https://classes.business.oregonstate.edu/reitsma/sprinkles.json.

Download the content of https://classes.business.oregonstate.edu/reitsma/sprinkles.json and store it in a file called *c:\temp\sprinkles.json*.[6]

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

First, for programmatically loading the data into MongoDB, copy the data loading code from the previous exercise (5.3). Then make the following changes:

6. 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 example, we will read the JSON from a local file and insert it into the database.

- Set the collection to `Sprinkles`
- Set `json_file` to `c:\\temp\\sprinkles.json`

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

Using *mongo.exe*, check to see if the data made it over:

use foo

db.Sprinkles.find()

Now the reverse: programmatically finding the number of sprinkles for which *Time.TotalMinutes* > 50. Again, we use pretty much the exact same program as we used for retrieving *swatches* data in *Exercise* 5.3, except for a few simple changes:

- Set the `collection` to `Sprinkles`
- Replace the query with the following:

```
result = db.Sprinkles.count_documents({"Time.TotalMinutes" : {"$gt" : 50}})
```

- Print the result as follows:

```
print(result)
```

The result of running the program should be 3 (Check this against the *sprinkles.json* file).

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 resources (TE 1.0), to a system running off of a NoSQL database with JSON representing resources (TE 2.0). Whereas the relational/XML model worked fine during the 13 years of TE 1.0, the new NoSQL/JSON alternative provides the advantage that JSON is used as both the resource 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 not very large, 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 one has data governance policies and practices in place which minimize the risk of data inconsistency, then one should at least consider a modern NoSQL database as a possible candidate for storing 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. Resource 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 resource 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 resource editor.

Authoring ≠ Editing

One way in which we can categorize digital libraries is by distinguishing 'content collections' from 'meta collections.' In a meta collection, no actual content is kept; only meta data —data about content— are kept. A good example of a meta collection is [NSDL.*org*](). NSDL (or National Science Digital Library) maintains meta data of about [35 digital library collections]() and allows searches over those 35 collections. The actual resources 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 resource they hold —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. Resources 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 resources 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 resource, 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 resource editors work with the authors to rework their documents so that they comply with the required structure. Once done, however, the resources are still in text-processed form. Hence, as we have learned in the

previous chapters, 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 TeachEngineering editing staff known internally as 'taggers.'[1]

TE 1.0 Tagging: Not Quite WYSIWYG XML

As discussed in the previous chapters, all TE 1.0 resources 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 resource content would almost certainly lead to difficulties. 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 resources 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 resources 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 resource documents without having to know their XML, yet Authentic would save their resource 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— it protected taggers from violating the Schema, thereby guaranteeing that documents remained valid.

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

***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 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 resource, the TE 1.0 system offered a (password-protected) web page where the tagger could test-render the resource.

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-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 resource 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 resource was the most labor-intensive step of the accessioning process, once we had an XML version of a TE resource, 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. **Resource check-in**. The tagger would check the resource into a central code repository (aka version control) system. Code repository systems such as Subversion 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 software source code, these systems can of course also be used for tracking and maintaining other types of electronic data sets, for instance XML or JSON files. Hence, in TE 1.0, taggers, once a resource had been converted to XML, checked that resource 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 resources 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 resource. 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 resource 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 resources. If it deemed a resource to be in violation of one or more of its rules, no meta data would be generated for it and the resource 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 resources. We named this process 'the spider' because its method of picking up resources 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 resource all the data it is looking for, after which it then looks for references or links to other resources and then crawls those in turn. Whereas most modern crawlers are multi-threaded; *i.e.*, they simultaneously crawl more than one resource, the TE 1.0 spider was simple and processed only one resource 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 twice a day. Figure 3 shows the process of 'spidering' a TE 1.0 resource; 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 the
                                     // 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 example (b).

4. **Document rendering**. One last step in the resource production chain in both TE 1.0 and 2.0 is actual rendering of a resource in users' web 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 resource indexing. Whereas all of the resource's content could be rendered directly from its XML, some aspects of rendering required a

database query. An example is a resource's 'Related Curriculum' (Figure 4). Whereas a resource 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 resource'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 resource's XML.

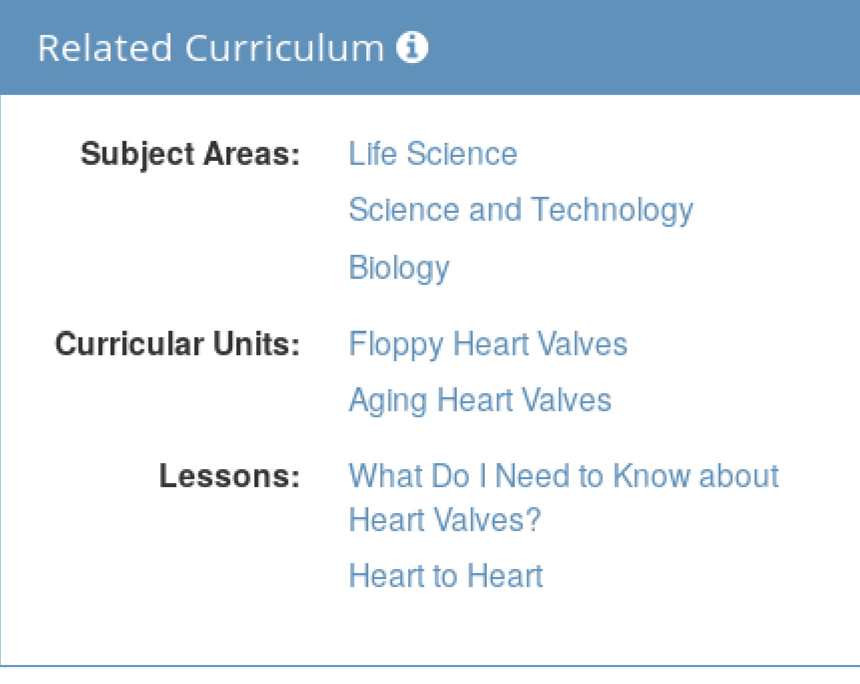


Figure 4: *TE 1.0 rendering of The Mighty Heart activity's 'Related Curriculum.'*

A second example of database-reliant resource rendering in TE 1.0 concerns a resource'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 resources is a so-called *many-to-many*

one (a standard can be related to multiple resources and one resource can have multiple standards), in TE 1.0 standards were stored uniquely in the database and resources 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 resource.

TE 2.0 Tagging: Much More WYSIWYG 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 resources in XML needed to be installed on each editor's computer along with the XML schema for each type of curriculum resource.

-The editing workflow had a number of steps that required the editor to understand specialized software, including *Authentic* and the *Subversion* version control system.

-Previewing a rendered resource 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 resources (and edits to existing resources) 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 taggers to add and edit documents from their web browser; no additional software required. As such, TE 2.0 includes a web/browser-based resource 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 *Tiny*MCE, 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.

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*

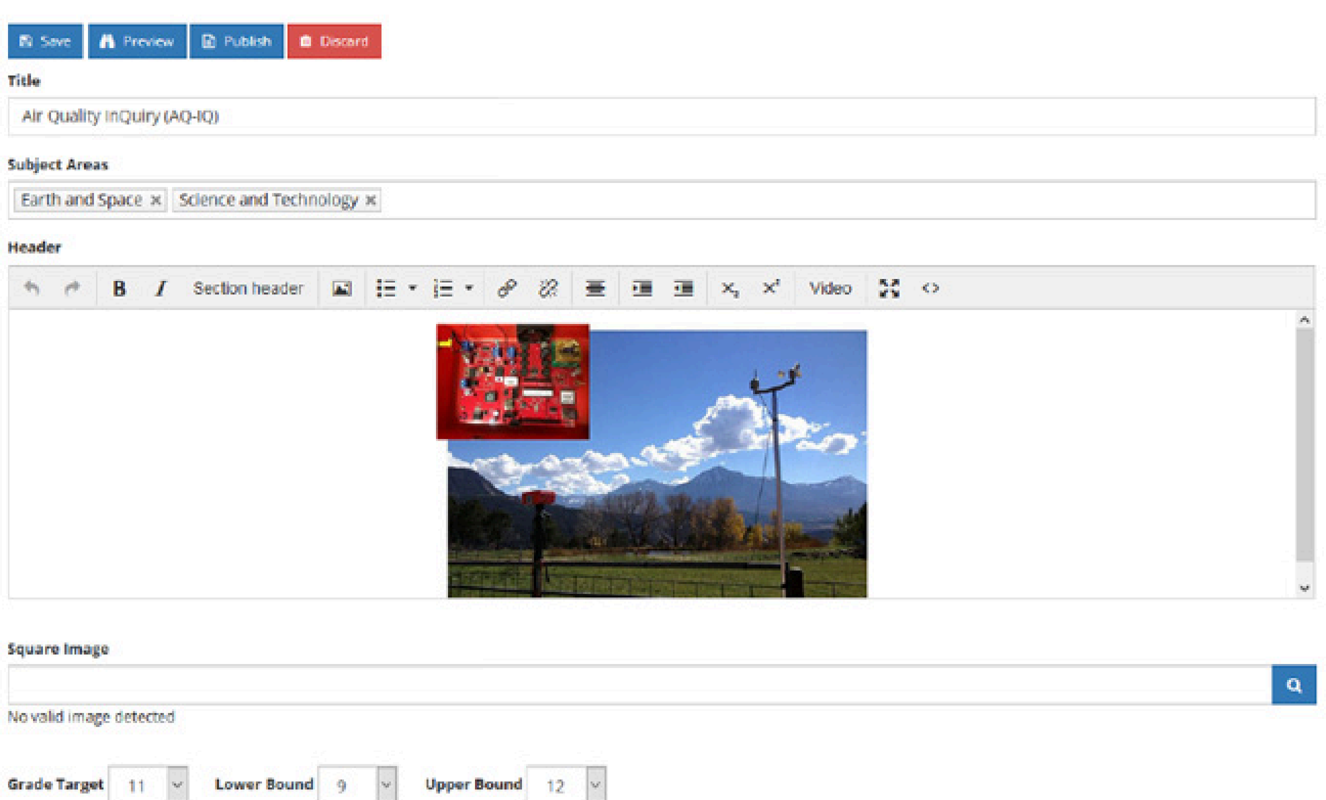


Figure 6: *TE 2.0 document editing interface for a Curricular Unit*

Figure 6 shows an example of editing a resource in TE 2.0. The interface provides a few options to support the editor's workflow. The *Save* button saves the in-progress resource to the (RavenDB) database. Resources that are in a draft state will not be visible to the public. The *Preview* button shows what the rendered version of the resource will look like to end users. The *Publish* button changes the resource's status from *draft* to *published*, making it publicly visible. Any errors in the resource, 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 resource which violate content rules is impossible.

As in the case of TE 1.0, resources in TE 2.0 are hierarchically organized in that resources 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 resources are stored on the file system (Figure 7).

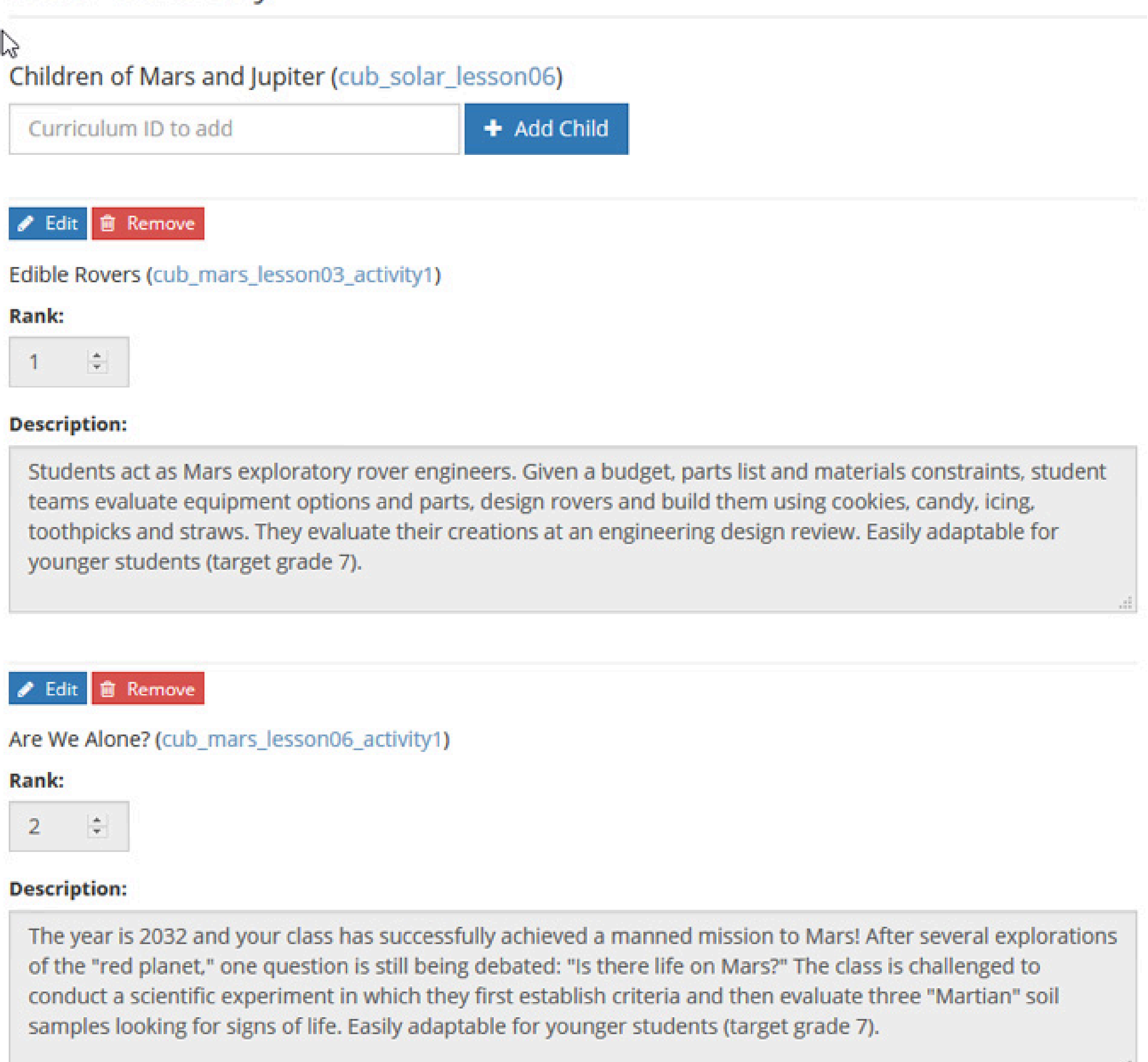


Figure 7: *TE 2.0's interface for specifying a resource's child documents*

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 resources using Authentic, and check the resulting XML resource into a version control system. With TE 2.0, editors edit resources using a familiar almost-WYSIWYG interface. The software behind the scenes takes care of the technical details of translating the resources into JSON and storing them in the RavenDB database.

TE 2.0 Document Ingestion and Rendering

With TE 2.0's architecture, the resource 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. **Resource check-in.** In TE 2.0, there is no resource check-in process; *i.e.*, no process of moving files from the local system into the TE repository of resource. When taggers save the resource, it is stored in RavenDB as JSON.
2. **Meta data generation**. In TE 2.0, there is no separate meta data generation process. As noted in chapter 4, TE 2.0 neither generates nor stores meta data. The JSON representation of the curriculum resource is the single version of the TE reality. Whereas TE 1.0 always generated and exposed its meta data 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 meta data harvesting protocols such as OAI-PMH (Open Archive Initiative–Protocol for Metadata Harvesting) have dwindled in popularity.
3. **Resource indexing.** There is no separate resource indexing step in TE 2.0. Since resources are saved directly to RavenDB, RavenDB will itself re-index its resources. Hence, there is no need to crawl and discover new or modified resources.
4. **Resource rendering**. At a high level, the resource 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 TE 1.0's PHP.
   Whereas in TE 1.0 the hierarchical relationships between any pair of resources were stored as parent-child rows in a relational database table, in TE 2.0, the relationship between all of the curriculum resources 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 resource'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 resources.

```
{ "CurriculumId": "cla_energyunit",
            "Title": "Energy Systems and Solutions",
            "Rank": null,
            "Description": null,
            "Collection": "CurricularUnit",
            "Children": [
                {
                    "CurriculumId": "cla_lesson1_energyproblem",
                    "Title": "The Energy Problem",
                    "Rank": 1,
                    "Description": null,
                    "Collection": "Lesson",
                    "Children": [
                        {
                            "CurriculumId":
```

```
                "cla_activity1_energy_intelligence",
              "Title": "Energy Intelligence Agency",
              "Rank": 1,
              "Description":
                "A short game in which students
                find energy facts among a variety
                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, curriculum resources in TE 1.0 only stored the identifiers of the standards to which the resource 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 resource. 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 resource which is aligned to that standard. In addition, storing the standards with the curriculum resource 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 resource-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 resource.

```
"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 particular
        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 resource 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 resource, updating the XML schema was the only step needed to allow taggers 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 software developer to make code changes to the edit interface. On balance, however, since resource schemas do not change that often, the advantages of a (much) more user-friendly resource editing experience outweigh the occasional need for code changes.

7. 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. For our purposes 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 always remains.[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.

1. In computer science, the application of so-called 'formal methods' is aimed at mathematically and logically analyzing and specifying code execution paths. This is in contrast to the more common way of testing code paths by submitting the code to a variety of pre-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.

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 maintained in a so-called code-repository or source-control or version-control system. These systems —examples are *Git*, *Subversion*, *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 rather 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, *etc*.

Version control 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 stored in 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 well. 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 such as 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' 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 —> Test site —> 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.

Table 1: TE 1.0's system monitoring processes

Monitor	*Frequency*	*Details*
'Systems up' test	Once per minute	A simple test to see if the TE 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, and 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.
Meta data harvesting checker	Once a month	A process which queries web 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 machine 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 July 2022, the TE 2.0 code base comprises 345 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 step 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 which uses the production database. Additionally, this build process ends with a series of 'smoke tests' which perform automated browser-based testing using Selenium, a tool for web browser scripting. These tests check key functionality of the TeachEngineering web site.
- 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

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.

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' (CI/CD). 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 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 its own source control system 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 managed by the cloud service provider. This is especially beneficial for small teams such as the TE team. Instead of worrying about things such as operating system updates and hardware maintenance, small teams can focus their resources on activities that make 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 subsequently travel drops to a level that does not require that 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 mouse 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 e-mail alerts in cases of adverse events such as unusually heavy CPU load, memory usage, excessive amounts of HTTP 5xx errors or slow site responses.

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 responses 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 three 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 one's system; *i.e.*, search engines such as Google, know about one's content and assess that content as an attractive target. This information obviously must come from the parties owning the search engines; it is not information internal to one's 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 one's 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 one's site is viewed by the world's most popular search engine.

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 resources the library has at any moment for any K-12 grade or grade range or how many different institutions have contributed resources and how many they have contributed.

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

8. A New Application: The NGSS Explorer

The NGSS as a Network

In the time elapsed between the first edition of this text (2017) and the current one (2022) numerous changes have been made to TeachEngineering (2.0). One of those is the addition of a new tool for exploring the Next Generation Science Standards; the NGSS *Explorer* tool.

At several locations in the previous chapters, we have referred to 'educational standards;' learning outcomes or 'performance expectations' that express knowledge and skills that USA K-12 students are meant to master as they progress through their education. In chapter 1 we mentioned that in the USA the formulation of these standards is the prerogative of the individual states. We also mentioned, that these standards are frequently updated and reformulated and that efforts have been made to harmonize standards between states. One of those efforts resulted in the *Next Generation Science Standards* or NGSS (NSTA, 2022); a set of K-12 science and engineering standards which at the time of this writing (July 2022) has been adopted by 20 states (*ibid.*)

The NGSS are constructed in a hierarchical, three-dimensional manner in that they are broken up into 12 so-called *Topics*. Examples of topics are *Energy*, *Engineering Design*, *Matter and its Interactions*, *etc*. Topics apply at one or more grade levels. For each topic, one or more *performance expectations* (PEs) exist. For instance, one of the PEs for the topic *Energy* at grade 5 is that students should be able to "*use models to describe that energy in animals' food (used for body repair, growth, motion, and to maintain body warmth) was once energy from the sun.*"

These PEs themselves are composites of three, more fundamental types of standards: *Crosscutting Concepts* (CCs) —ideas and concepts that apply across disciplines; *Disciplinary Core Ideas* (DCIs) —the core ideas and concepts of a discipline such as Earth Sciences or Biology, and *Science and Engineering Practices* (SEPs) —practices shared across all of science and engineering.

Adding all Topics, PEs, CCs, DCIs and SEPs present in the NGSS, we arrive at a total of 845 standards. These 845 standards maintain a total of 2,145 relationships with each other.

The traditional way of viewing the NGSS is through a set of linked tables. Figure 1 (on-line at https://www.nextgenscience.org/dci-arrangement/k-ess2-earths-systems) shows an example of this.

Students who demonstrate understanding can:

K-ESS2-1. Use and share observations of local weather conditions to describe patterns over time. [Clarification Statement: Examples of qualitative observations could include descriptions of the weather (such as sunny, cloudy, rainy, and warm); examples of quantitative observations could include numbers of sunny, windy, and rainy days in a month. Examples of patterns could include that it is usually cooler in the morning than in the afternoon and the number of sunny days versus cloudy days in different months.] [*Assessment Boundary: Assessment of quantitative observations limited to whole numbers and relative measures such as warmer/cooler.*]

K-ESS2-2. Construct an argument supported by evidence for how plants and animals (including humans) can change the environment to meet their needs. [Clarification Statement: Examples of plants and animals changing their environment could include a squirrel digs in the ground to hide its food and tree roots can break concrete.]

The performance expectations above were developed using the following elements from the NRC document *A Framework for K-12 Science Education*:

Science and Engineering Practices	Disciplinary Core Ideas	Crosscutting Concepts
Analyzing and Interpreting Data Analyzing data in K–2 builds on prior experiences and progresses to collecting, recording, and sharing observations. • Use observations (firsthand or from media) to describe patterns in the natural world in order to answer scientific questions. (K-ESS2-1) **Engaging in Argument from Evidence** Engaging in argument from evidence in K–2 builds on prior experiences and progresses to comparing ideas and representations about the natural and designed world(s). • Construct an argument with evidence to support a claim. (K-ESS2-2) - ***Connections to Nature of Science*** **Science Knowledge is Based on Empirical Evidence** • Scientists look for patterns and order when making observations about the world. (K-ESS2-1)	**ESS2.D: Weather and Climate** • Weather is the combination of sunlight, wind, snow or rain, and temperature in a particular region at a particular time. People measure these conditions to describe and record the weather and to notice patterns over time. (K-ESS2-1) **ESS2.E: Biogeology** • Plants and animals can change their environment. (K-ESS2-2) **ESS3.C: Human Impacts on Earth Systems** • Things that people do to live comfortably can affect the world around them. But they can make choices that reduce their impacts on the land, water, air, and other living things. *(secondary to K-ESS2-2)*	**Patterns** • Patterns in the natural world can be observed, used to describe phenomena, and used as evidence. (K-ESS2-1) **Systems and System Models** • Systems in the natural and designed world have parts that work together. (K-ESS2-2)

Connections to other DCIs in kindergarten: N/A

Articulation of DCIs across grade-levels:
2.ESS2.A (K-ESS2-1); **3.ESS2.D** (K-ESS2-1); **4.ESS2.A** (K-ESS2-1); **4.ESS2.E** (K-ESS2-2); **5.ESS2.A** (K-ESS2-2)

Figure 1: *NGSS Topic Earth's Systems at grade K as a table of PEs and linked standards (The* ***bolded*** *standard codes in the Articulation section, are links to tables representing those standards)*

There is little doubt that viewing the NGSS through a set of linked tables is easier than following the connections through a single text. However, the notion that we have 845 standards maintaining 2,000+ connections with each other seems to invite an alternative representation, namely that of a network of nodes (the standards) and links between those nodes. This is what the NGSS Explorer in TeachEngineering does. Figure 2, for instance, shows the same '*Earth's Systems* at grade K' standard as Figure 1, but this time in the form of a network. The network is interactive in that users can can navigate through it by interacting with the nodes and the texts of the standards, by selecting different groups of standards, depth levels, graph display options, *etc*.

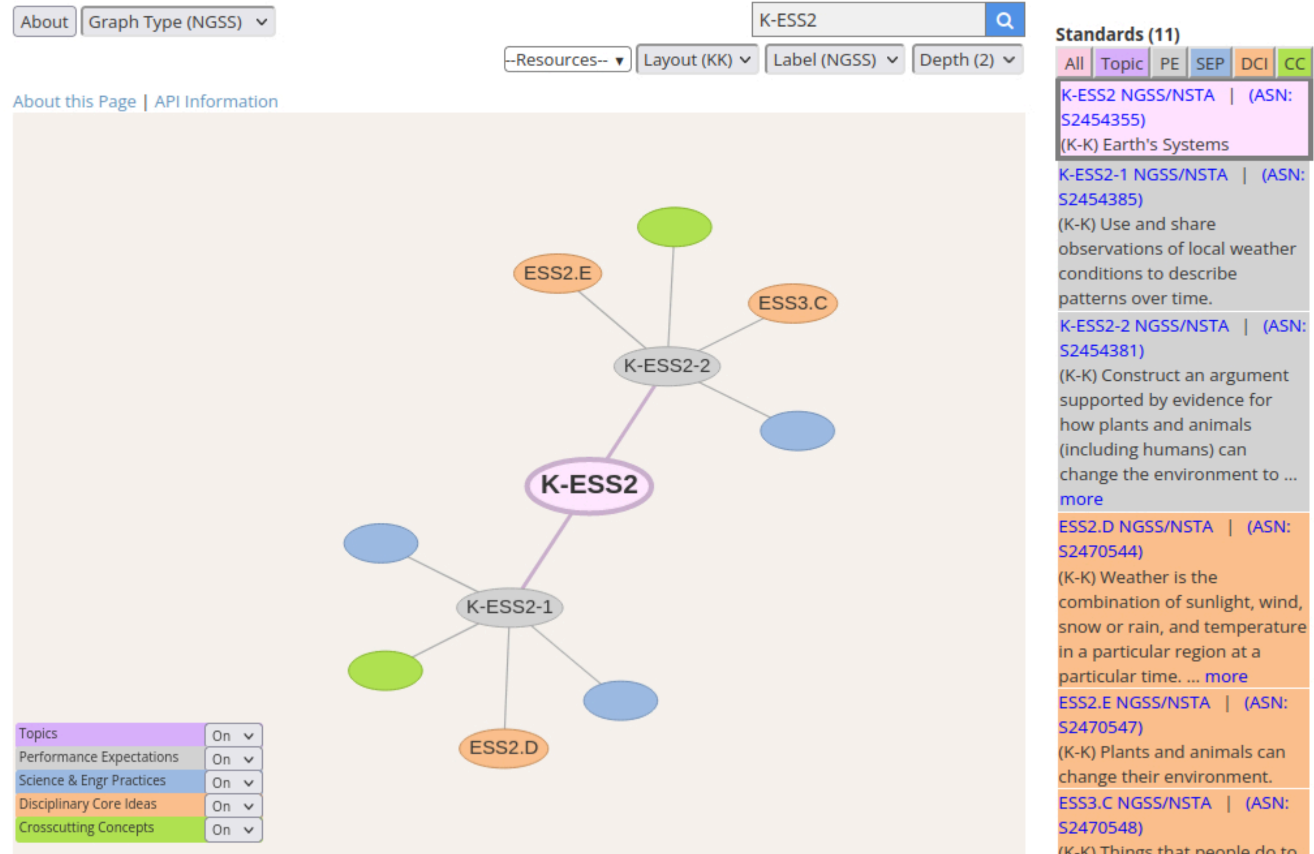


Figure 2: *NGSS standards as a network.*

Similarly, the network can be viewed at different *depths*. With 'depth' we mean the distance from the displayed standard (node) expressed as the number of links away from that standard. For instance, the depth of the network of Figure 2 is 2; *i.e.*, from standard *K-ESS2*, we follow up to two link steps. Figure 3 shows the network for that same standard *K-ESS2*, but this time with a depth of 3.

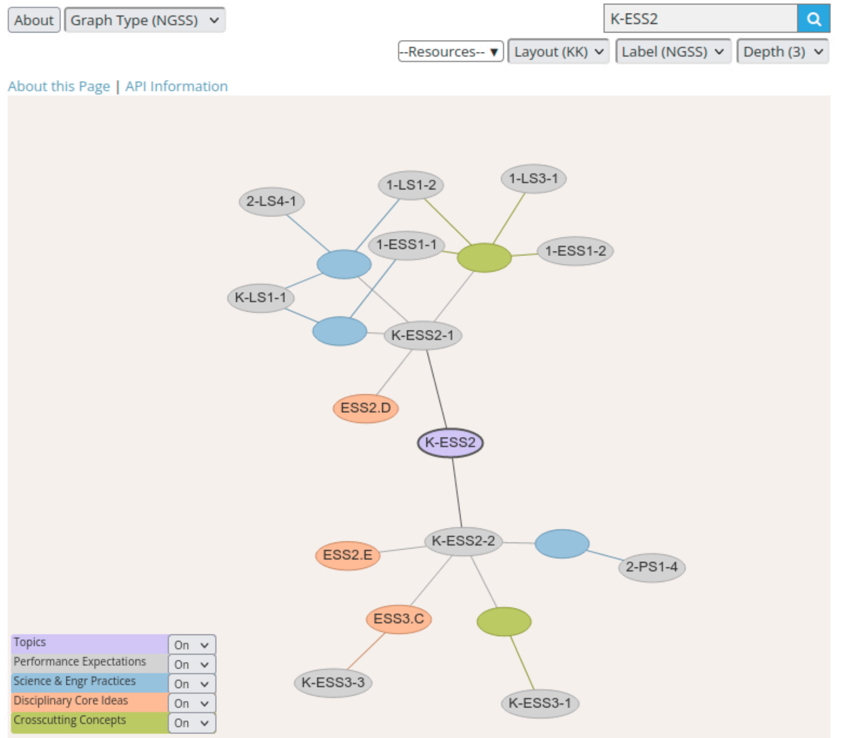


Figure 3: *NGSS standards as a network with Depth = 2*

One of the advantages of networks is that they can be extended by adding new nodes and connections. One application of that is to include learning resources which are aligned to those standards. Figure 4, for instance, shows not only the same standards as displayed in Figure 2, but also a series of aligned K-12 learning resources which support teaching those standards: one from TeachEngineering (the star), four from Mystery Science (the grey squares) and four from a collection of NGSS-endorsed resources (the blue-grey diamonds).

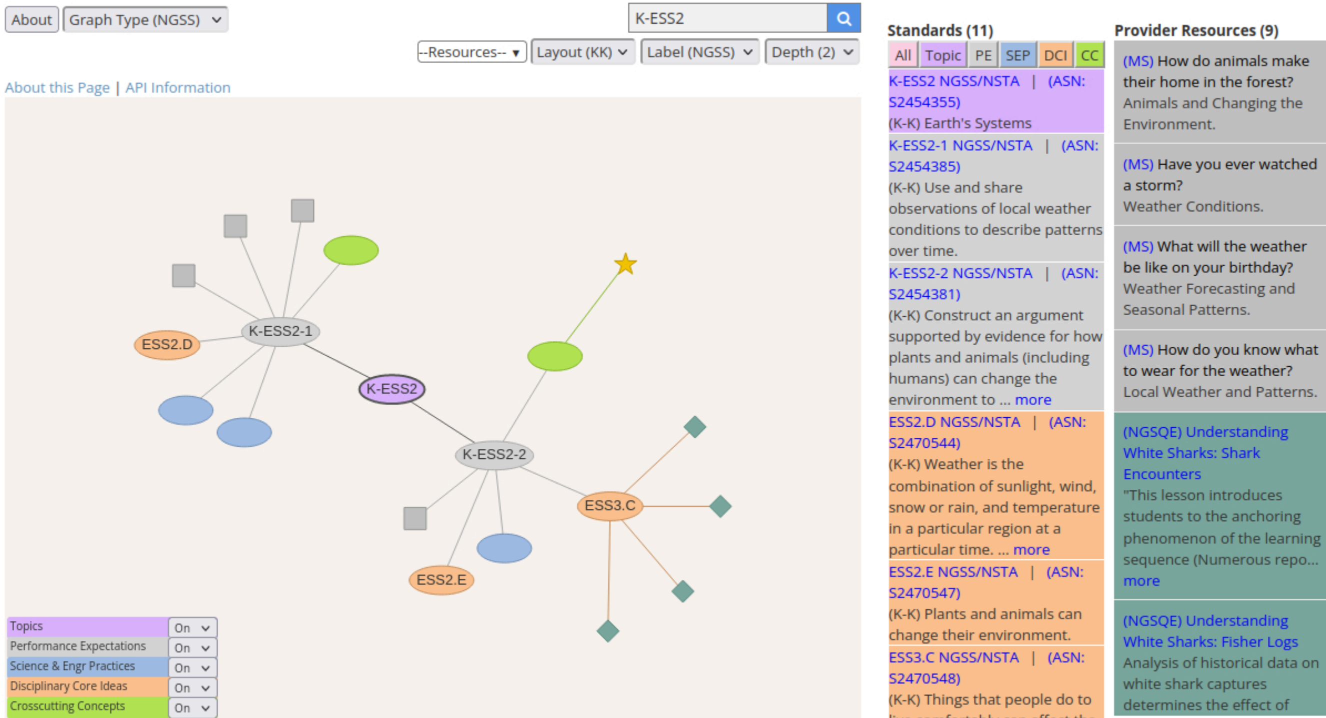


Figure 4: *NGSS standards with aligned learning resources.*

We invite readers to explore what else you can do at https://www.teachengineering.org/ngss_explorer. If you find things not to be working as expected, do not hesitate to drop an email at *reitsmar 'at' oregonstate.edu*.

How Does All This Work?

The technology that runs the NGSS Explorer is the same technology that runs the rest of TE 2.0; *i.e.*, a RavenDB (NoSQL JSON) database, C#, HTTP and a lot of JavaScript. The only other component is a set of R libraries that is used to compute the positions of network nodes before rendering that network on the screen. The network rendering itself and the direct user interaction with it such as zooming, selecting and moving network nodes are all done by the *vis.js* JavaScript library.

Figure 5 shows the basic architecture of the NGSS Explorer.

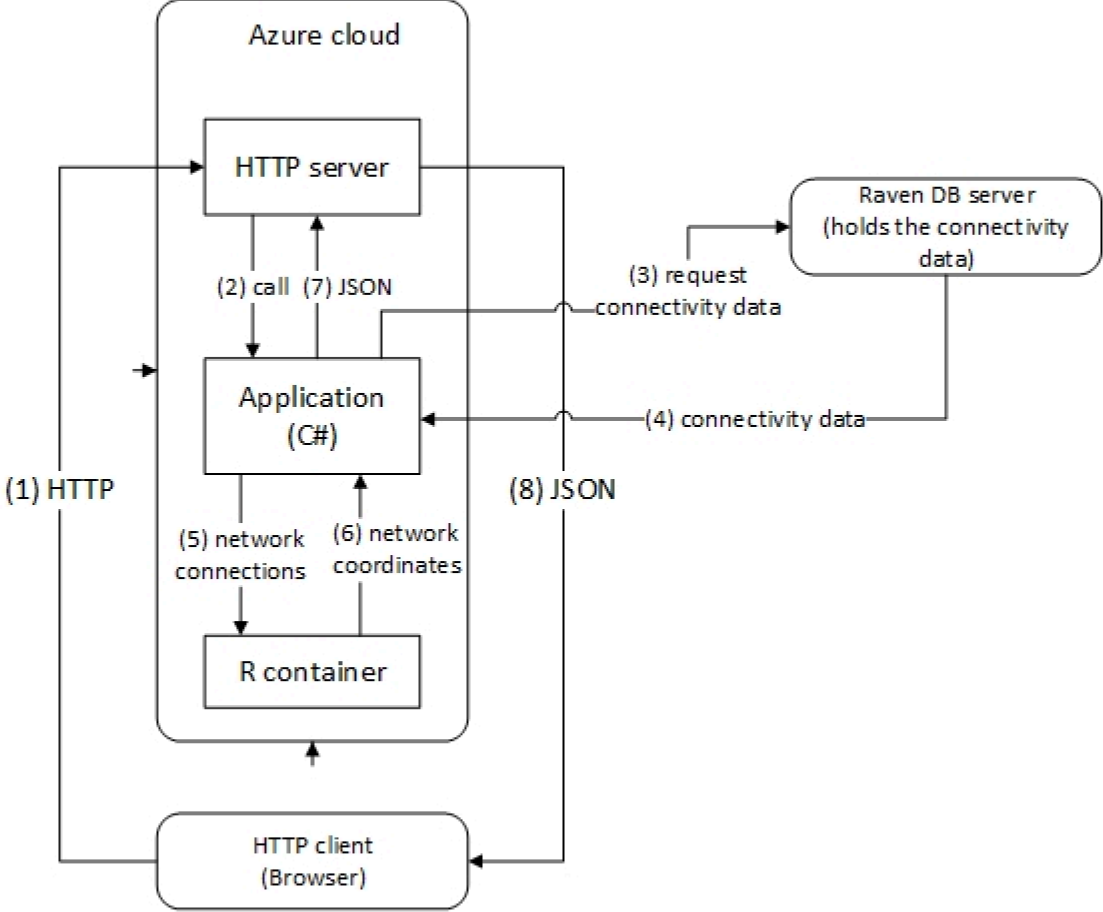


Figure 5: *NGSS Explorer architecture*

The RavenDB database holds all the data of the NGSS standards, their connections, their alignments with teaching resources, and basic information about those resources, such as who publishes them, their title and summary and —if available— their location on the web.

The NGSS explorer essentially runs in the user's web browser as a JavaScript program. When a user starts it (Step 1 in figure 4), it reaches out to the TeachEngineering HTTP (web) server with a request for a complete network; *i.e.*, a request for a data structure representing any and all nodes with any and all possible connections between those nodes (Steps 2, 3 and 4, 7 and 8 in Figure 4.). Since any and all possible networks that can be displayed each represent only a subset of this complete network, this data structure is loaded only once during an NGSS Explorer session; it is held in memory by the JavaScript program running in the user's web browser.

Once a user submits a request for a specific network, the JavaScript running in the browser extracts from its complete network only those nodes and connections which are implied by the user's request.

At that point only two more steps remain: 1. figuring out where —at which locations— on the screen to display each of the nodes and 2. display the nodes at those locations and all connections between them.

Of these, the first —figuring out the positions of all the nodes to be displayed— is the trickiest one. Many algorithms for solving this problem exist; see McGuffin (2012) for a good introduction. The NGSS Explorer offers two of these: Kamada-Kawai (1989) and Fruchterman-Rheingold (1991). Kamada-Kawai is used as the

default.
Several implementations for both of these algorithms are available on the Internet/web. We found that the ones implemented in R's *igraph* library worked really well, so those are the ones used for the NGSS Explorer.

Figure 5 once again shows how this functionality is integrated into the system. Once a user requests a specific network and the JavaScript program running in the browser has determined which nodes and connections are involved, it submits a request to the TeachEngineering HTTP server for the positional information of the network's nodes (Step 1 in Figure 4). The HTTP server forwards this request to a C# program (Step 2), which in its turn calls R (step 5). R then computes the positional information and returns it to the C# program (Step 6), which in its turn returns it as a JSON structure to the HTTP server (Step 7), which returns that structure to the user's browser (Step 8). Once the JavaScript program has received the nodes' positional information, it passes that and the connectivity data to vis.js which then displays the network in the browser.

References

Fruchterman T.M.J., Reingold E M. (1991) Graph drawing by force-directed placement. *Software: Practice and Experience*, 21, 1129-1164.

Kamada T, Kawai S. (1989) An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31, 7-15.

McGuffin, M.J. (2012) Simple Algorithms for Network Visualization: A Tutorial. TSINGHUA SCIENCE AND TECHNOLOGY, ISSN 1007-0214 02/11. 383-398. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6297585 (accessed, July 7, 2022)

NSTA — National Science Teachers Organization (2022) *About the Next Generation Science Standards*. https://ngss.nsta.org/About.aspx (accessed, July 7, 2022)

On several occasions in this text you will encounter exercises which require you to create files containing some textual content; perhaps a Python 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]

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.

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
```

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 write the file 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*) as well as 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 11,000+ 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: 1. store text and 2. 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 bytes than those needed for the 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*, *VS Code* 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.

Note: most word processing programs allow you to save text in **.txt* (text) format; *i.e.*, as unformatted text. Hence, if you insist on using word processing software for coding always save your file in *.txt* format (Note that you may have to rename the file after saving it to adjust the file's file extension).

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.

2. Files with unformatted text are known as '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 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 recommend that you do 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*.

The following table should help you find a few suitable, free-of-charge 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-of-charge text editors

	Notepad++	*bluefish*	*TextMate*	*vi*	*emacs*	*nano*	*Visual Studio Code*
Windows	x	x		x	x		x
MacOS		x	x	x	x		x
Linux		x		x	x	x	x

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.

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 response times as well as database query times had all greatly increased. For individual users accessing the system, response times had increased 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 server 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 from 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 https://gsuite.tools/traceroute) 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 had TeachEngineering reply to the requests with an empty response; *i.e.*, no content, but 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.

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 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*

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.

(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.

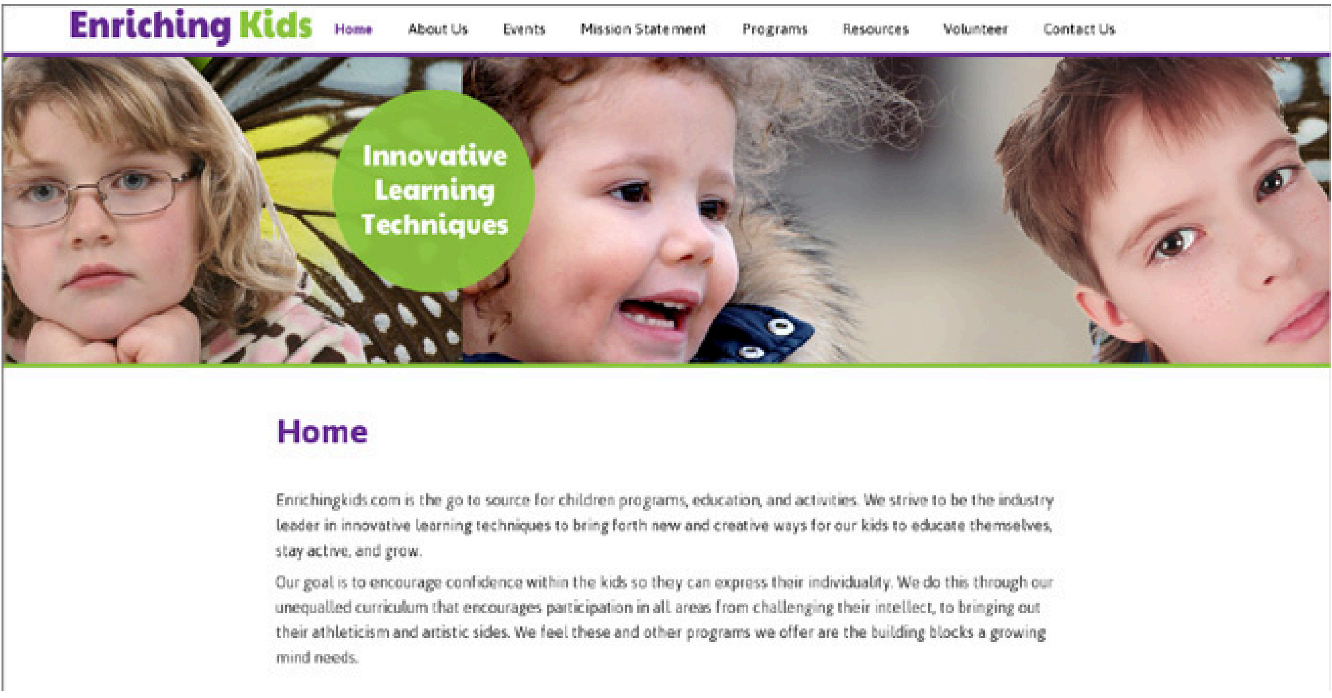


Figure 1: *Home page of enrichingkids.com.*

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.

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

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.

Let us take a look at [TE 2.0's *robots.txt* file]().

```
# This is the production robots.txt file
User-agent: *
Crawl-delay: 2
Disallow: /standards/browse
Disallow: /curriculum/print/*
Disallow: /view_lesson.php?*
Disallow: /view_curricularunit.php?*
Disallow: /view_activity.php?*
Disallow: /search_results.php?*
Sitemap: https://www.teachengineering.org/sitemap.xml
```

The items other than the *User-agent* and *Crawl-delay* items, need some explanation. For instance, why are there references to PHP pages; *e.g.*, *view_lesson.php* or *view_activity.php*, especially since in previous chapters we stated that whereas TE 1.0 was written in PHP, TE 2.0 does not have any PHP (it is all in C#)?

The reason for these odd references and for requesting robots not to crawl them, is that when TE 2.0 was launched, it was —and still is— very likely that other websites contained links to these TE 1.0 pages. Hence, we kept these pages functional in that they can be requested but those requests are then redirected to the TE 2.0 version of them. This is an important point. Far too often do we see websites make URL changes which render all their old URLs invalid, thereby invalidating any and all the links to the old pages maintained by the rest of the world. A far better and much friendlier way of handling URL migration is to redirect the requests for the old URLs to the new URLs; at least for some reasonable amount of time, so that the linking websites have some time to reset their links.

To see this URL redirecting at work, pull up the *https://www.teachengineering.org/sitemap.xml* page in your web browser (this may take few seconds), and click on one of the follow-up **.xml* links. Note how this results in a long list of TE 2.0 URL pages.

Keeping those old URLs up, however, implies a real risk that robots continue to crawl them, which is what we do not want. Therefore, whereas the old URLs continue to function, the *robot.txt* file instructs the robots not to crawl/use them (*Disallow*:). Interestingly, no *Disallow* directive is specified for *https://www.teachengineering.org/sitemap.xml*. The reason, of course, is that following the links on that page eventually leads to TE 2.0 URLs, so there is no problem with letting robots crawl that page.

Detecting Robots

Robots also can be a drain on data communications, data retrieval and data storage bandwidth. About 2/3 of all HTTP requests TeachEngineering 1.0 received originated 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 future.

Real-time Robot Detection

Several methods for real-time robot detection 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.
- **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 code 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 detecting out returning robots. Blacklists can be built up internally. For instance, one of the products of periodic in-house after-the-fact robot detection 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 detection.

After-the-fact Robot Detection

Just as for real-time robot detection, 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

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:

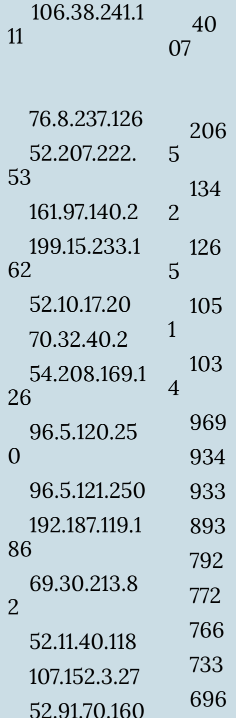
  ```
  select count(distinct(host_ip))
  from april_2016_hits;
  ```

  296,738

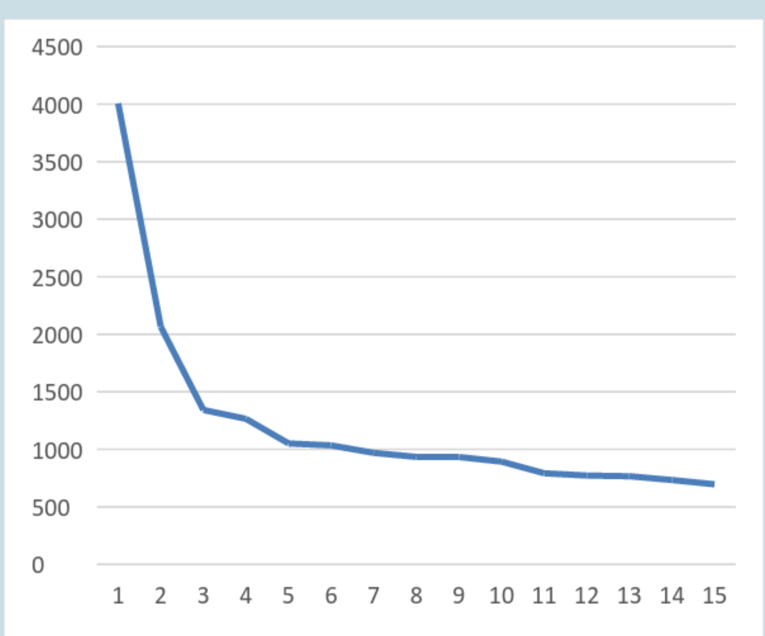
- Find the 15 largest hitters and their hit counts in descending order of hit count:

  ```
  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 their hit counts.



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://whatismyipaddress.com/ip-lookup):
  - 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 seconds (70, 140, 210 and 280 seconds).

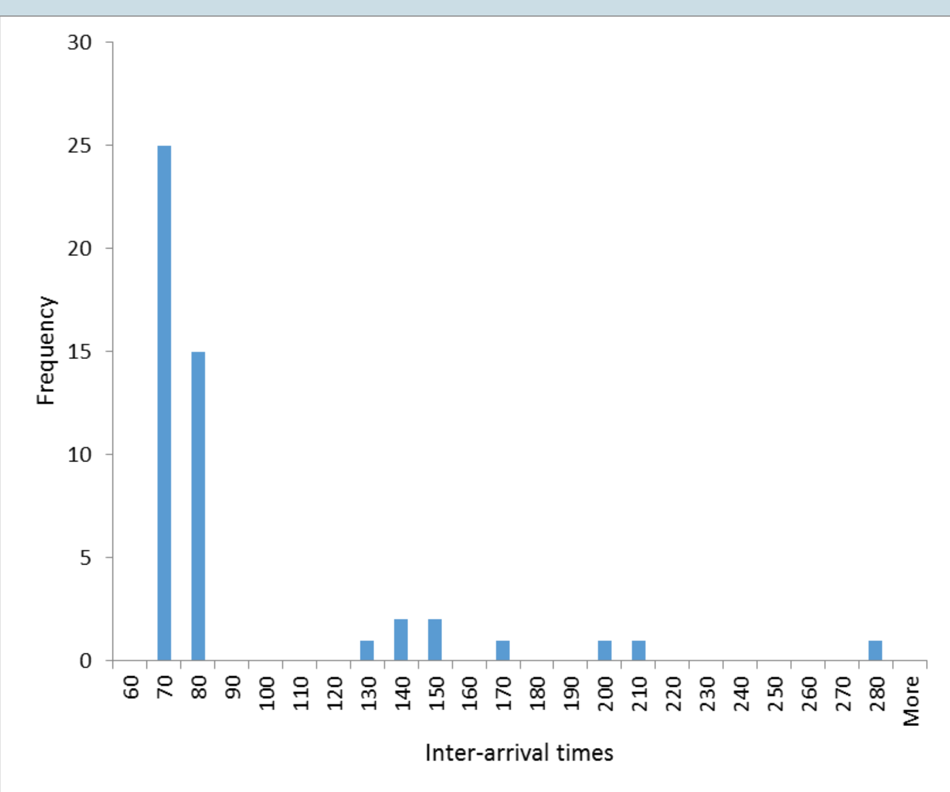


Figure 2: *histogram of the inter-arrival times for the first 100 hits by IP 106.38.241.111.*

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, it is reasonable to 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_hits
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% (1,436 + 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 1,000+ 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.

This work is licensed by René Reitsma and Kevin Krueger 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.

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.

This page provides a record of changes made to this publication. Each set of edits is acknowledged with a 0.01 increase in the version number. The exported files, available on the homepage, 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
1.0	**09/01/2022**	**Publication**	