modelica.university: A Platform for Interactive Modelica Content

Michael M. Tiller[1] Dietmar Winkler[2]

[1]Xogeny, USA, michael.tiller@xogeny.com
[2]University College of Southeast Norway, dietmar.winkler@usn.no

Abstract

The World Wide Web was conceived of as a medium for the expression and exploration of scientific and engineering ideas. However, much of the innovation in web technologies is now focused on consumer facing applications. Although science and engineering content is available on the web (Wolfram Alpha, 2017), there are not that many tools that allow engineers and scientists to create and build scientific and engineering applications.

Fundamentally, HTML and HTTP are certainly sufficient for the creation of scientific and engineering content just as they are for the creation of online magazines and websites. But while a number of "content management systems" have been created to facilitate the publication of prose, there are very few such tools that cater to making it easy to create scientific and engineering content.

In this paper, we will present a platform which can be thought of as a content management system for scientific and engineering content. We will start by describing what we believe to be the fundamental requirements for such a system. From there, we will discuss two different applications built on this platform. The first is an interactive tutorial for teaching the basics of the Modelica languages and the other is an example application that involves creating interactive content for use in an engineering course on hydro-electric power generation. This content will be published on the modelica.university domain and we are already collaborating with others to contribute additional content to the site.

Keywords: Modelica, web, cloud, education, content management

1 Introduction

1.1 Background

The initial goal of this project was to recreate a previous application entitled "Tour of Modelica" using a newer platform for deploying web-based engineering tools and content. The previous version of the application was written to provide a "tool free" experience for learning the basics of Modelica. Similar efforts involving the OpenModelica tool OMNotebook have also been undertaken (Palanisamy et al., 2016).

Because the tutorial was web-based, it could be used as part of an interactive, introductory tutorial at events like the North American Modelica Users' Group meetings without requiring participants to install tools. Furthermore, the only prerequisite was a browser. So, the tutorial was not just tool neutral, but OS neutral as well. During live events, the tutorial material was used by participants running Windows, MacOS and even iOS.

However, the tutorial was based on older infrastructure and the decision was made to upgrade the tutorial. At the same time, it was also decided to make the underlying platform available for others to create web-based educational content based on Modelica. The domain name modelica.university was registered for this new site.

1.2 Requirements

The underlying platform was created to support the creation of web-based engineering analysis tools. Many lessons from the creation of proprietary tools were factored into the design of the infrastructure that supports the deployment of these applications. In this section, some high level requirements for the platform (based largely on the experience of developing earlier tools) will be enumerated.

1.2.1 Hypermedia

The success of the web is, in part, due to the ability of hypertext to link together content from different sources. For most users and developers of web content, this is most typically associated with HTML (W3C, 2016).

However, it should be noted that the concept of hypertext has since been generalized to the more general term "hypermedia". The concept of hypermedia extends the idea of describing links and relationships not just between text and content within that text, but to data in general. In hypermedia, a URL is used to refer to a "resource". Those resources represent data of some kind and may have potentially multiple different potential representations (*e.g.,* an image resource could be represented as either a JPG or a GIF image). This modern conception of hypermedia and the use of hypermedia as an architectural style for building network based applications was formalized in (Fielding, 2000).

But in order to support this, formats besides HTML are required. This is because HTML is focused on being a declarative way to represent documents (hence the presence of elements like <img> (image), <h1> (header)

and <p> (paragraph). But in order to generalize the approach to data, a whole range of new formats like HAL (Kelly, 2016), Collection+JSON (Amundsen, 2013) and Siren (Swiber, 2016) were developed.

The most essential aspect of these formats is that they allow generalized data (in most cases serialized as either XML (Maler et al., 2008) or, more commonly, JSON (ECMA International, 2011)) to express hypermedia concepts like relationships to other resources and/or actions that can be performed on these hypermedia resources.

At the dawn of the World Wide Web, hypermedia was recognized as an essential component for the expression and exploration of scientific and engineering ideas. Our experience shows that the power of applying hypermedia concepts to science and engineering is still not fully realized and our goal was to not only include it as a requirement for managing scientific and engineering content, but to exploit it even further than most existing platforms.

1.2.2 API

Nearly all web applications require some kind of API to interact with. Generally speaking, the two main functions of an API are to provide information and the carry out tasks. The term "Command Query Responsibility Segregation" (CQRS) refers to an architectural style where these two responsibilities are clearly and cleanly delineated (Fowler, 2011).

As such, it is no surprise that our API requires both of these functions. An API is generally just the "middle man" between the client (*e.g.,* the web application) and one or more sources of information leveraged by the server (*e.g.,* databases, file systems). The *query* functionality allows the web application to request information from those sources via the API. The *command* functionality allows the web application to request tasks to be performed by the server. The main difference between the command and query functionality is that queries are, generally speaking, idempotent, *i.e.,* they don not change the state of the server while the command functionality typically exists solely for the purpose of mutating the server side state. Furthermore, querying functionality generally relies on caching as an optimization to speed up the fetching of information and to ensure its "freshness" while commands frequently invalidate caches as a result of mutation.

For our purposes, we need querying functionality to provide us with text, images, models, simulation results, *etc.*. We need the command functionality mainly to request computational tasks like simulations and optimizations to be performed.

1.2.3 Content Creation

A significant impediment to web and cloud adoption in the world of science and engineering is the fact that there is not much overlap in technical skills between engineers and web developers. As such, engineers need to rely on web developers to help them with creation of web based tools. Of course, HTML is relatively easy. But to move beyond simple static markup requires a wider range of skills. Unfortunately, people with those skills tend to be drawn to more "consumer oriented" projects with the potential to reach very large markets (social networking, advertising, search engines, games, *etc.*). As a result, the rate of innovation and adoption in the engineering sector has traditionally been and continues to be slow.

In order to break this cycle, it is essential to develop technologies that make it easy to turn people with specialized scientific or engineering skills into content creators. Of course, this is nothing new. But, again, many of the development resources are focused on empowering broader sections of society and less on science and engineering.

In reducing the learning curve for non-experts, there are two important aspects to consider. The first is easing the creation of content. This means being able to easily make scientific and engineering content accessible through the APIs, *e.g.,* connecting the API to existing data sources or computational capabilities. The other aspect is the visualization of the underlying content in the web browser. For the purposes of this project, we require that both of these are facilitated to some extent.

1.2.4 Third Party Tools

While `modelica.university` is being hosted publicly, the infrastructure it is build on was developed to support proprietary tools and applications. Many of those applications are intended to be hosted on private networks. It is quite common that customers insist that all data remain on private networks. In those cases, it is impossible to rely on third party services hosted on the public Internet (*e.g.,* Amazon EC2, Google Cloud Platform, Digital Ocean).

So none of the software libraries used by the `modelica.university` infrastructure rely on services that are hosted exclusively on the public Internet. However the requirement to avoid public services was relaxed for this project to make deployment easier and more cost effective.

1.2.5 Job Processing

In our earlier discussion on APIs, we mentioned the need to perform "computational tasks". But for scalability reasons, it is frequently important to delegate these computational tasks away from the API server. Without such delegation, the response of the API server itself could be slowed down considerably by CPU intensive tasks running on the same machine. Furthermore, numerical tools are often written in languages like FORTRAN, C++, Python, Julia, *etc.*, while web servers, databases and other backend services are written in languages like Javascript, Java and so on. To address both the scalability and interoperability, it is often convenient to introduce message queues or worker queues. These provide a way to link together various services in a scalable way while avoiding the tendency toward monolithic architectures. The term "microservices" (Susan Fowler, 2016) refers to an architectural style which is very much aligned to these require-

DOI 10.3384/ecp17132725

ments.

2 Content Management Platform

Now that we have elaborated some of the requirements for the application, we will quickly review how we have addressed those requirements in our implementation.

2.1 Backend

The term "backend" refers to aspects of the application not handled in the web browser. This includes the web server that serves the application, databases, authentication, "memcache", *etc.*.

2.1.1 API

We start our discussion of the backend with the API itself. For `modelica.university`, we leverage the Heisman API framework. Heisman is a proprietary framework developed by Xogeny for creating hypermedia APIs. The main feature of this framework is the ability to define so-called "resources" using an intrinsically hypermedia-oriented structure. Once defined, an HTTP based API can automatically be synthesized for those resources. The emphasis on hypermedia semantics means that resources are able to easily express not just data about themselves but also relations to other resources as well as actions that can be performed by resources.

The fact that an HTTP based API can be automatically synthesized is important because it avoids having to write a great deal of boilerplate code to handle pedantic HTTP specific details like status codes, caching, etags, accept header processing and so on.

We have taken an "API first" approach to application development. As we will discuss shortly, once the resources are defined and the API is automatically generated, a generic API browsing application is already available for the API.

2.1.2 Resources

The resource oriented approach to application development means that resources need to be defined with hypermedia semantics in mind. Our definition of resources is largely inspired by the Siren hypermedia format. Specifically, a resource is described by three distinct types of information.

The first type of information a resource can provide is the "properties" of the resource. This is the true data associated with the resource. For example, if the resource represents results from a time-domain simulation, the "properties" might be the values of the independent and dependent variables.

The second type of information a resource can provide about itself is metadata. The metadata for a resource includes a textual description of the resource as well as zero or more textual "classes" that identify (in some domain specific way) what the resource represents. For example, if the resource represented simulation results, the set of textual classes might include the string `"simulation_result"`. It may also include the name of a more specialized class, *e.g.,* a resource might include `"drive_cycle_result"` and `"simulation_result"` where the former is a specialized form of the latter.

The final, and arguably most important, type of information associated with a resource is "links", which convey how one resource relates to other resources. The ability to "link" to other resources is the essence of hypermedia. The link between resources is always associated with one or more "relations". Relations, like classes, are typically domain specific names although the Internet Assigned Numbers Authority (IANA) has defined a collection of standard link relations (Internet Assigned Number Authority, 2017). For example, the `item` relation is used to define the relationship between a (collection) resource and any other resource "contained" in it. Similarly, the `collection` relation may appear on each item resource to link back to the enclosing container resource.

2.1.3 Domain Specific Resources

The term "resource" is an abstraction used to refer to any kind of data that might be accessed over a network. To help understand what a resource is and how they relate to our application, we will provide several concrete examples for discussion in this section.

Static Content A very common type of resource is a file. In fact, web servers like the Apache or NGINX web servers treat files precisely as hypermedia resources by providing a way to refer to those files as network addressable streams of bytes. Heisman also provides a means to serve files as network addressable resources. However, in our application the contents of the file are only part of the resource. We also allow the metadata and link information to be associated with a file. Just by associating such information with the files, it becomes possible to quickly and easily define a rich range of structural information about the resources associated with an application. This hypermedia oriented information can be supplied within the file itself (by serializing it as a Siren instance) or programmatically via special handler routines registered with the server that add hypermedia annotations to those files.

This ability to annotate files with hypermedia information means that much of the content being managed by the content management system can be represented by files that are statically served directly from a file system. This capability is important because it helps us address the requirement that creation of content should be easy and intuitive for people who are not programmers or web developers. Using this functionality, much of the application can be built simply by dragging and dropping files into directories. We will demonstrate this further in the context of both applications discussed later. It is worth noting that content served from the filesystem is also much easier to version control vs. content stored in a database.

Dynamic Content In addition to static content, most applications depend on the ability to create and manipulate data dynamically in response to user actions. For example, each time a simulation is performed we might wish to store those simulation results away for retrieval later. In some cases, we might want a resource to represent a very specific type of data (*e.g.,* simulations performed by a given user) with specific fields (*e.g.,* model simulated, user who requested the simulation, time request was made, time required to complete the simulation). In other cases, we might require a way to create, manipulate and query arbitrary (schema free) data. While the former often requires specialized resources to be created, Heisman provides a standard collection resource to handle the latter.

Job Brokers The final resource type used in these applications is essential for handling requests for computational work. In both applications, the computational work required is running simulations. Because nearly every scientific or engineering application will require one or more types of computationally intensive analyses, Heisman includes already implemented resources called "job brokers". These job brokers provide an API for requesting work to be done, tracking the status of that work and reporting back the successful result or an error message. The code is independent of the task to be performed. This means that a job broker can be easily created and associated with one or more specific computational tasks required by the application.

The hypermedia semantics allow us to cross reference job requests with job results. In other words, for a given simulation result we can follow the links associated with that result to find the original request and vice-versa. Such cross referencing of resources can be used for traceability and to determine provenance of data.

2.2 Communication

The capabilities described so far rely on several different communication mechanisms. In this section we will quickly summarize each of these.

The web application running in the browser relies on hypertext transfer protocol (HTTP) (Fielding et al., 1999) for invoking queries and commands. These HTTP requests are received and acted upon by code on the server that maps these requests to the underlying resources referenced in the requests.

The "job broker" resource uses a tool called Redis (Sanfilippo and Noordhuis, 2017) to implement message and worker queues. It is via Redis that messages are sent between the API server and the workers that perform any CPU intensive computations.

2.3 Deployment

Desktop tools are typically compiled into binaries and distributed via "installers". In contrast, web applications are deployed (often, continuously) to servers where they can then be accessed via a web browser. This simplifies the install process for the user (since they only have to enter a URL in a web browser), but the process of deploying software to these servers safely and efficiently adds a whole new dimension to the software development process[1]

An important technology for the deployment of network services is called "Docker". Technically, Docker is a tool designed to make it easy to access the special Linux process groups called "containers". But this explanation does not adequately explain Docker's role or capabilities.

Conceptually, Docker is a technology for creating extremely resource efficient virtual (Linux) machines. The efficiency comes from Docker's use of kernel level features in Linux that isolate groups of processes while allowing them to share large amounts of read only data in memory and/or on the file system.

The backend server for `modelica.university` is a Node (Node.js Foundation, 2017) application written in TypeScript (Microsoft, 2017). To generate a Docker image, the `dockergen` Node package (Tiller, 2017) is used. The `dockergen` script creates a `Dockerfile` which specifies how the application should be packaged for deployment to a Docker host. Once a Docker image is built, it can be run as a container on a Docker host. Since this is a public application, we can take advantage of commercial Docker hosting services.

The actual application is made up of several distinct Docker images executed using the "compose" functionality of Docker. In addition to the API server image, the backend consists of several other images. One image runs the Redis server. Another image runs a NGINX web server to act as a reverse proxy. A third image runs the API server. The final image executes the workers for the computational tasks processed via the worker queue. With Docker, it is quite simple to activate multiple containers running the worker image. This allows us to easily scale up the number of workers during periods of high load. Another advantage of Docker that all the machines in a cluster are securely firewalled within the same network. Only ports that have been explicitly opened to machines within the cluster are accessible outside the cluster.

3 Application 1: Tour of Modelica

3.1 Objective

Now that we have discussed how the underlying infrastructure is implemented, let us get into the details of the first application. As mentioned previously, the "Tour of Modelica" application is a reimplementation of an earlier web application. The application is structured in the form of chapters and lessons. In each lesson, the user is presented with some introductory material about a specific aspect of the Modelica language and starting from some sample code is asked to carry out several modeling tasks. After completing the exercises, the user moves on to the next lesson and/or chapter.

[1] So much so, that the term "DevOps" was coined to refer to the combined set of development and operational skill required to deploy web applications.

DOI 10.3384/ecp17132725

To complete each task, the user must be able to edit, compile and simulate Modelica code. The code editing is done in the browser, but the compilation and simulation is requested via the API and performed by a worker that uses OpenModelica (Open Source Modelica Consortium, 2016) to compile and simulate each model.

3.2 Content

The content for the application consists primarily of lessons, chapters, lesson text and sample models. All of these can be represented as static resources using the functionality previously discussed in 2.1.3.

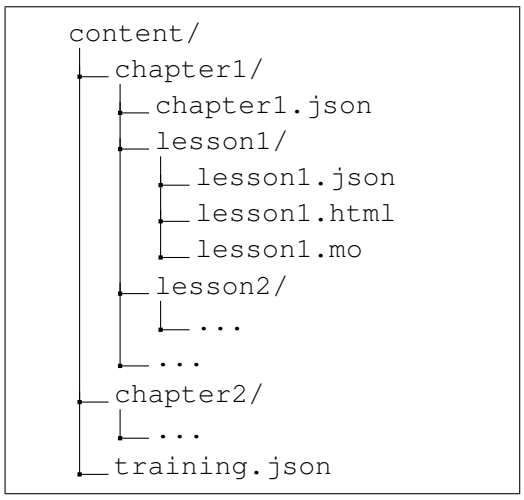

Figure 1. Fragment of the files system.

A fragment of the file system content is shown in Figure 1. All content is rooted in a directory named `content`. The files are organized by chapters and lessons although this is strictly a convention. Files ending in the `.json` suffix are interpreted as hypermedia resource descriptions. These JSON files contain the metadata, properties and links discussed previously. Let us look at the `lesson1.json` file to as an example of how one such resource might be described:

```
{
  "title": "Simplest Model",
  "properties": {}
  "class": ["lesson", "start"],
  "links": [
    { "rel": ["text"],
      "href": "./lesson1.html" },
    { "rel": ["source"],
      "href": "./lesson1.mo" },
    { "rel": ["task"],
      "href": "resource://simulate" },
    { "rel": ["chapter"],
      "href": "../chapter1.json" },
    { "rel": ["training"],
      "href": "../../training.json" }
  ],
  "query": {
    "rel": {
      "training/*": { "embed": true },
      "chapter/*": { "embed": false },
      "source/data": { "embed": false },
      "text/data": { "embed": false },
      "task": { "embed": true }
    }
  },
}
```

From this description, we can see that this resource is titled "Simplest Model" and has no properties. Because this resource is a lesson, we include the `lesson` class in its description. It also has the `start` class which we can use in our application to locate the first lesson. The `links` section provides (respectively) links to the HTML markup for the lesson text, the initial model source, the job broker that will run the simulation, the chapter that this lesson belongs to and the `training.json` file which describes all the chapters that are part of the "Tour of Modelica" application. The `query` section describes what information about the resource should be returned from each HTTP request[2]. By default, all resources have a "default query" that describes what information about that resource is to be returned for each HTTP request. The `query` section here is defining the default query. Note that clients (*e.g.,* our web application) are free to specify their own query with each request. In this way they can request more or less information to be provided, depending on their needs.

This is a lot of information. Furthermore, nearly all of it is essentially repeated from one lesson to the next where only a few details are changed. Fortunately, Heisman provides a way for us to programmatically augment the contents of resources represented by files on the file system. In this way, we are able to write code to automatically fill in all the information based conventions like the directory structure or the lesson name. In fact, the only thing we cannot figure out automatically is the title. As a result, the task of creating a new lesson resource becomes as easy as creating a file that contains:

```
{
    "title": "Simplest Model"
}
```

A similar process is used to augment information about other types of content on the file system (*e.g.,* chapters). This relatively small amount of upfront work to define specialized handlers greatly simplifies the process of content creation and making the process accessible to non-programmers. In addition, allowing data to describe its relationship to other data means that that information and logic does not need to be coded into the client. This makes development of the client easier and more general.

3.3 Visualization

3.3.1 Generic Browser

There are many aspects about the operation of a web browser that most users are not aware of. One of those

[2] In our API, the primary response content type is Siren. Because Siren allows related resources to be embedded in a response or simply linked to, our query format must specify which approach to use for each matching resource. Hence the `embed` field.

aspects is the `Accept` header. This is a header included with an HTTP request that lets the server know what types of content it expects back. The default `Accept` header for Google Chrome looks like this:

```
Accept: text/html,application/xhtml+xml,
application/xml;q=0.9,image/webp,*/*;q=0.8
```

This is essentially a list of content types the browser understands. But it also defines the clients order of preference for the different content types. The `Accept` header is useful to the server because it is possible that a given resource could be represented in multiple formats and the `Accept` header provides a clue as to which format is preferred.

The `Accept` header is important in API development because it can be used to determine whether the request that the API is handling is coming from a browser or from Javascript code. If our server sees that the request is for HTML, it will respond to the request by serving up a page that loads an embedded browser application. That web application is actually a generic graphical user interface for Siren APIs that comes bundled with the server. We will talk about the user interface application in greater detail shortly.

This is part of the "API first" philosophy discussed earlier. As a result of following this philosophy, every API developed in this way automatically comes with a graphical user interface. Furthermore, remember that Heisman automatically synthesizes an HTTP API based on the resources that are registered with it. What this means, in practice, is that once you describe your resources, *you immediately and automatically get both an HTTP API and a web application.*

3.3.2 Custom Visuals

As mentioned previously, Simran is the web application that is launched when browsing the API. Simran is a proprietary technology used by Xogeny to create web based UIs for scientific and engineering applications.

Simran is really a browser running in a (web) browser. Generally speaking, web browsers like Chrome or Firefox are used for browsing HTML or other widely used content types. If you are a scientist or engineer, the problem is that web browsers do not understand more technical formats (*e.g.,* Modelica models, `.mat` files, FMUs).

The API browsing application compensates for this by providing a web application that is extensible. Because the browser application is built around the notion of hypermedia (primarily in the form of Siren representations) and not hypertext (*i.e.,* HTML), we can represent many different content types *and* the relationships between them. In a sense, this is a lower level alternative to HTML.

That, by itself, may not sound that useful. But it becomes more useful because of the plugin system. Via the plugin API, it is possible to extend the browsing application with any number of specialized visual components. While the base browser application is a generic browser that renders all Siren resources essentially the same, when enhanced via plugins the browser application is able to provide custom rendering for different content types based on the metadata, properties or relations of the resource.

For example, using just the base browser, our "Tour of Modelica" application is shown in Figure 2.

There we can see the first lesson and its related resources rendered using metadata. Furthermore, we can click on links to follow the various resources. But each resource will be visualized in the same generic way. However, after we provide a plugin with custom visuals for lessons and chapters, putting the **same URL** in our web browser will yield a rendering of the lesson like the one shown in Figure 3.

The plugin system is based on React (Facebook, 2017). Normally, each React component independently specifies what "properties" it understands when instantiating a component. We turn this around a bit and standardizes these properties to conform to a canonical representation of a hypermedia resource. As a result, all React components are "equivalent" in the sense that they are instantiated with the same set of properties but with different values. But, through the plugin system we have the freedom to customize *which* component to use for each hypermedia resource. In this way, we are essentially creating a browser that can easily be extended to understand any kind of scientific or engineering content instead of being limited to just those standardized in the HTML specification by the W3C.

In the case of the "Tour of Modelica" site, the plugin defines custom renderers for lessons, chapters and the training overview. In addition, it leverages some standard and easily reusable visuals provided by the built in browser for applications and application suites.

For each application, the application developer can decide what types of content the browser should be capable of understanding and then simply add those visuals to their plugin. This modular approach to visualization makes it very easy to create a custom user interface for a particular domain and/or reuse components developed for other applications.

The authors would like to acknowledge the contribution of the `moijs` project for providing syntax highlighting and checking for the embedded Modelica editor as well as the CodeMirror project (Haverbeke, 2017) for the editor widget itself.

3.3.3 Mobile

Consumers of web applications and web content are increasingly consuming this content from mobile devices. Support for phones and tablets mainly involves making sure that layout of content makes sense for small form factor screens. In some cases, some content may be hidden on small displays. With `modelica.university` we have made every effort to support mobile devices.

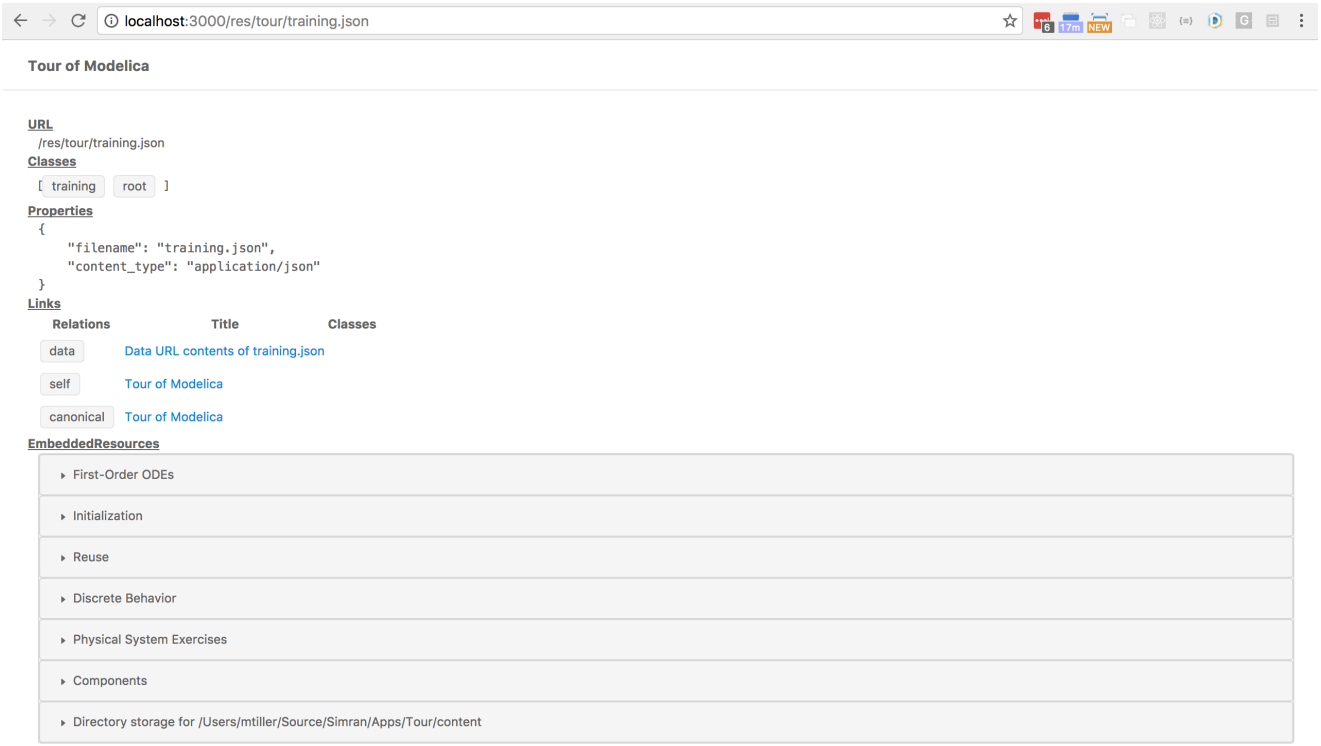

Figure 2. Generic rendering of Tour resources

4 Application 2: Hydro-Electric Power System

4.1 Objective

The second application example is a student exercise that is part of the Master's course "Object-oriented Modelling of Hydro Power Systems" at University College of Southeast Norway. The course starts with an introduction to the fundamentals of Modelica. Later on it moves on to model specific parts of a hydro-electric power system.

Typical modeling problems are:

- Waterway configuration
- Water hammer investigations
- Droop control behavior of the turbine governor

Being able to solve such problems interactively using only the browser as a tool without having to immediately understand Modelica code improves the physical understanding of the system. Once the physical understanding is there, creating more complex models and scenarios is easier for the students to achieve.

4.2 Content

The contents of this application are the different main problems and each with multiple configurations. For example, for the *Waterway* application different examples with a number of interconnecting pipes are given where the levels of the pipe ends need to be verified and checked that they make sense. This is sometimes not as easy as it sounds since pipes might connect to reservoir models which have a different height reference. So the student is given a set of parameters for the different pipe segments of other components of the water way and has to determine if the setup "makes physical sense".

For the *Water hammer* problem, one can investigate the influence of closing time of a valve depending on the pipe diameters and flow rates. The content would also provide certain restrictions like allowable maximum pressure in the pipes.

The *Droop control* (Wikipedia, 2017) problem contains data that describes the droop settings of one or more turbine controllers and lets one investigate the respective frequency dependent power productions.

The typical data structure of the content is shown in Figure 4.

4.3 Visualization

The real benefit for the second application will be the visualizations of the problems and especially solutions.

The *Waterway* problem is much more intuitively solvable when the students is presented with a sketch of the physical setup of the different pipe levels and other waterway components. Here the student can at once see a possible flow in the parameter set.

For the *Water hammer* problem a different method of visualization can be used. For example interactively showing unsuitable closing times by emphasising the pressure plots of setups that violate the restrictions. As the student changes parameters live (*e.g.,* via a slider), they get the plot

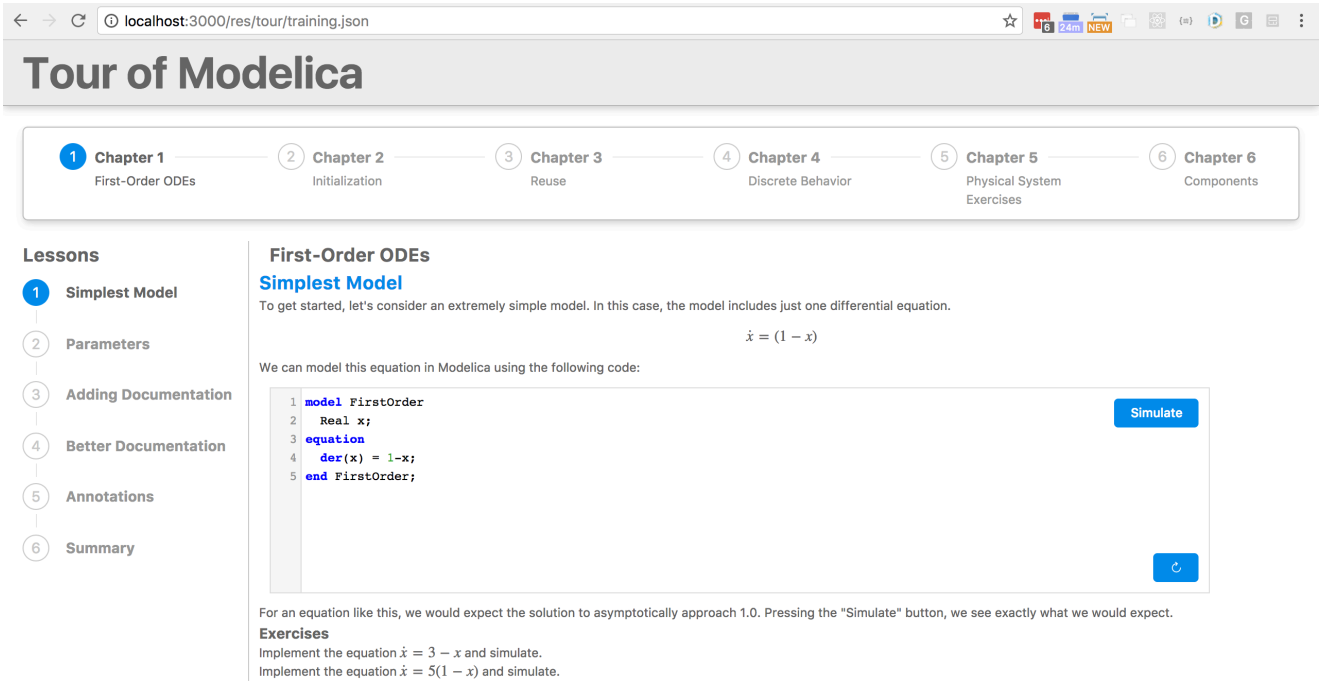

Figure 3. Custom rendering of Tour resources

results presented live based on a real simulation done in the background. The executed models can be supplied as Modelica source files or FMUs.

The *Droop control* problem can be visualized by providing interactive droop setting behaviours including limits and again reacting on parameters that can be interactively set.

Figure 5 shows a typical plot of the power sharing behavior of three generators with different droop settings.

5 Related Efforts

The pace of innovation in the web development landscape is breathtaking. It is nearly impossible to keep track of all the new technologies that emerge almost on a daily basis. The authors drew inspiration from many amazing projects, including:

- **Jupyter** A tool for interactive data science and scientific computing across all programming languages (Project Jupyter, 2017)
- **Nextjournal** - An interactive writing and programming environment for every stage of research from experimentation to publication (Nextjournal, 2017)
- **"What Can a Technologist Do About Climate Change? (A Personal View)"** - Bret Victor's sprawling essay on technologies that can help address climate change (Victor, 2015).
- **Modelica in Action** - An interactive notebook for compiling and simulating Modelica (Bonvini, 2017).
- **Modelica by Example** - An interactive book about Modelica (Tiller, 2016).

6 Conclusions

By leveraging the power of hypermedia and a wide array of open source technologies, we were able to build the `modelica.university` site and our two sample applications. We gained several insights as a result of this work.

6.1 Middleware

Creating a site like this involves creation of the underlying content, implementation of the necessary analysis capabilities, an HTTP API and a domain specific web application to support user interaction. But most of the domain specific work here is at the edges, *i.e.,* content creation and visualization. Through their API synthesis and browser architectures, the Heisman and Simran packages allow development resources to remain focused on those domain specific edges. This adds efficiency to the development process while providing a tremendous amount of reusability. Together, these two packages form the foundation of Xogeny's *Aperion* platform.

6.2 Current Status

At this point, `modelica.university` implements the two applications described in this paper. Our experiences

DOI 10.3384/ecp17132725

```
content/
├── waterway/
│   ├── waterway.json
│   ├── setup1/
│   │   ├── setup1.json
│   │   ├── setup1.html
│   │   └── setup1.mo
│   ├── setup2/
│   │   └── ...
│   └── ...
├── waterhammer/
│   ├── waterhammer.json
│   ├── long-ww/
│   │   ├── long-ww.json
│   │   ├── long-ww.html
│   │   └── long-ww.fmu
│   └── ...
├── droopcontrol
│   └── ...
└── hydro-power.json
```

Figure 4. Fragment of the files system.

with these applications further reinforces the importance of the requirements outlined at this start of this paper. We are confident that with each additional application, the platform will gain more and more capability as a browser for scientific and engineering content.

6.3 Future Plans

In terms of content, we hope that others will contribute more content in diverse subject areas to help us further validate our approach, refine our requirements and, ultimately, provide meaningful educational content for science and engineering students.

As for the platform, we feel its further development will be largely driven by use cases involving `modelica.university` and other proprietary projects. Now that the basic pieces of the architecture are implemented, there are countless optimizations we would like to make to improve responsiveness. There are also many types of content we would like to provide custom visualizations for (*e.g.,* time series data, version trees, diagram authoring).

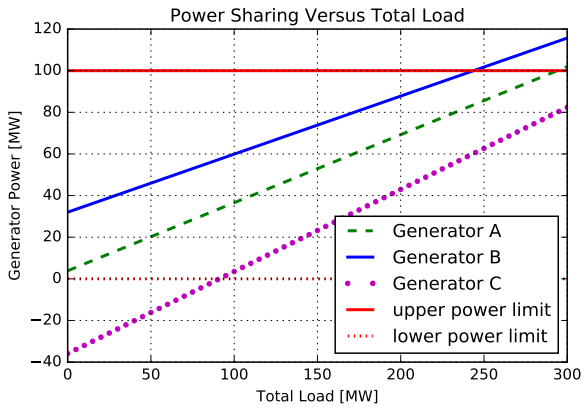

Figure 5. Example of a droop control visualization

References

Michael Amundsen. Collection+JSON - Hypermedia Type, 2013. URL `http://amundsen.com/media-types/collection/`.

Marco Bonvini. Modelica in action: compile and simulate models, 2017. URL `http://marcobonvini.com/modelica/2017/01/02/modelica-in-action.html`.

ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011. URL `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

Facebook. React - v15.4.2, 2017. URL `https://facebook.github.io/react/`.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999. URL `https://tools.ietf.org/html/rfc2616`.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

Martin Fowler. CQRS, 2011. URL `https://martinfowler.com/bliki/CQRS.html`.

Marijn Haverbeke. CodeMirror, 2017. URL `https://codemirror.net/`.

Internet Assigned Number Authority. About Us, 2017. URL `http://www.iana.org/about`.

Michael Kelly. JSON Hypertext Application Language, 2016. URL `https://tools.ietf.org/html/draft-kelly-json-hal-08`.

Eve Maler, Tim Bray, Jean Paoli, François Yergeau, and Michael Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

Microsoft. TypeScript - Javascript that scales, 2017. URL `https://www.typescriptlang.org/`.

Nextjournal. Nextjournal, 2017. URL `https://nextjournal.com/`.

Node.js Foundation. About Node.js, 2017. URL `https://nodejs.org/en/about/`.

Open Source Modelica Consortium. Openmodelica, December 2016. URL `https://openmodelica.org/`.

A. Palanisamy, M. Sjölund, and P. Fritzson. Generating OpenModelica Web Books Including Mathematical Typesetting from OMNotebooks, 2016. URL http://www.modprod.liu.se/filarkiv/1.672879/OpenModelica2016-talk15-Arunkumar-GeneratingOpenModelicaWebbook.pdf.

Project Jupyter. Project Jupyter, 2017. URL http://jupyter.org/.

Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2017. URL https://redis.io/.

Susan Fowler. *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*. December 2016. URL http://shop.oreilly.com/product/0636920053675.do.

Kevin Swiber. Siren: a hypermedia specification for representing entities, 2016. URL https://github.com/kevinswiber/siren.

Michael M. Tiller. Modelica by Example, 2016. URL http://book.xogeny.com/.

Michael M. Tiller. Generate a Dockerfile for any NodeJS application, 2017. URL https://www.npmjs.com/package/dockergen.

Bret Victor. What Can a Technologist Do About Climate Change? (A Personal View), 2015. URL http://worrydream.com/ClimateChange/.

W3C. HTML 5.1, 2016. URL https://www.w3.org/TR/html/.

Wikipedia. Droop speed control, 2017. URL https://en.wikipedia.org/wiki/Droop_speed_control.

Wolfram Alpha. Wolfram Alpha, 2017. URL https://www.wolframalpha.com/web-apps/.

 DOI 10.3384/ecp17132725