OpenDSA Documentation

Release 0.4.1

OpenDSA Project Contributors

November 07, 2016

CHAPTER 1

Introduction

This documentation describes various components of the OpenDSA system. This includes the support infrastructure for authoring OpenDSA modules, information helpful for developing AVs and exercises, documentation for the textbook configuration system (to control which modules and exercises are generated for a specific book instance and how the exercises are scored), documentation for using the front-end (client-side) portion of the logging and scoring infrastructure, and documentation for the back-end (server-side) API and data storage system.

AVs and exercises are typically built using the JavaScript Algorithm Visuailzation library (JSAV). Complete documentation for JSAV can be found at http://jsav.io.

If you are new to OpenDSA, it will help you to keep in mind as you look through this manual that it targets a lot of different things, but you are probably concerned with only one of several possible roles at any given time. Those roles include:

1. Content developer: Someone who wants to create or modify modules, visualizations, or exercises If so, you want to read the sections on creating content.
2. Instructor: Someone with a class to manage. If so, you want to read about the instructor tools. If you want to set up your own book instance by picking and choosing from the OpenDSA collection of materials, then look at compiling a book instance and the configuration system.
3. System administrator: Someone who wants to set up their own copy of OpenDSA. OpenDSA separates the front end content delivery server from the back end data collection server. You might want to set up either or both, in which case you should look at the installation guides.
4. OpenDSA Infrastructure developer: If you want to help to extend the OpenDSA infrastructure, then you will need to understand the details of the part of the system that you are targetting.

CHAPTER 2

Getting Started

2.1 Overview and Developer's First Steps

OpenDSA consists of content delivered by servers. Content is delivered in the form of "book instances", which are created by the *configuration process*. A book instance is accessed through a Learning Management System (at the moment, we are only supporting Canvas), with the files delivered by an LTI Content Provider. Various support operations are conducted by the OpenDSA Server. If you want to develop content, then create a book instance and view it, then you will need to set up the necessary infrastructure. For testing purposes, this has all been packaged together to simplify setting up a development environment. See https://github.com/OpenDSA/OpenDSA-DevStack for how to set this up.

Once you have the development environment in place, the next step is to get an account on a Canvas server. You can either use one provided by your institution, set up your own Canvas server, or use the public test server provided by Instructure at https://canvas.instructure.com. With your account in place, you can tell Canvas to create a course. The place to start is to create a course named "Test". You will then go back to your development environment, and create a *course configuration file*. You should start with one named "Test_LMSconf.json", made by copying the template in the config directory. You can then go to the top of the OpenDSA repository, and do `make Test`. If everything worked right, then you will have populated your course on Canvas with some content. At this point, you are ready to learn about the parts of the system that you need to know in detail so that you can do useful work.

2.2 Project Communications

The primary discussion forum for topics related to OpenDSA and JSAV is our Piazza forum at https://piazza.com/class/i1v25wdagpr6sn.

Issues (bug reports and suggestions) related to any of the repositories should be posted to their respective GitHub issue trackers.

2.3 Repositories and Official Mirrors

Main development is done out of repositories hosted at GitHub. We use a GitHub "organization" at https://github.com/OpenDSA. Here is a list of the individual repositories that we use:

- The main OpenDSA development repository: https://github.com/OpenDSA/OpenDSA. The stable releases are kept in a separate repository at https://github.com/OpenDSA/OpenDSA-stable.
- Most developers should use the version of JSAV distributed with OpenDSA. However, if your task requires the most recent changes then the development version of JSAV can be found at: https://github.com/vkaravir/JSAV.

- We use the Khan Academy infrastructure for exercises and distribute the necessary portions with OpenDSA.
- Support for setting up OpenDSA servers can be found at https://github.com/OpenDSA/OpenDSA-DevStack.
- The OpenPOP project is in a separate repository at https://github.com/OpenDSA/OpenPOP.
- The QBank project is in a separate repository at https://github.com/OpenDSA/QBank.

The stable releases of OpenDSA and JSAV are mirrored at: http://algoviz.org/OpenDSA/ and http://algoviz.org/OpenDSA/JSAV, respectively. The built version of the stable modules are mirrored at: http://algoviz.org/OpenDSA/Books. The development versions of OpenDSA and JSAV are mirrored at: http://algoviz.org/OpenDSA/dev/OpenDSA and http://algoviz.org/OpenDSA/dev/OpenDSA/JSAV, respectively.

2.4 JSAV

Visualizations are developed using the JSAV (JavaScript Algorithm Visualization) library. Documentation for the JSAV API can be found at: http://jsav.io/

2.5 File Structure and File Naming Conventions

The following refers to the OpenDSA content or “client side” repositories (“OpenDSA” and “OpenDSA-stable”).

Content materials come in the form of modules (in RST), exercises, AVs, etc. There are various top-level directories, as explained below and in more detail in the project README file (https://github.com/OpenDSA/OpenDSA/blob/master/README.md). Within the RST, AV, SourceCode and Exercises directories, the materials are subdivided into subdirectories based on topical content (such as Sorting). These content subdirectories are mirrored across all of the materials subtypes. That is, if there is a Sorting subdirectory in the AVs directory, there should also be one in the RST directory, Exercises directory and the SourceCode directory to match. In addition, each of the major top-level directories contains a subdirectory named Development. All content starts life in the Development subdirectory. Once it is completed, polished, validated, and had a thorough code review, Dr. Shaffer will move code out of the Development subdirectory to an appropriate content subdirectory.

Algorithm visualizations, proficiency exercises, and related code live in the AV directory.

Exercises built using the Khan Academy exercise infrastructure lives in the Exercises directory.

Tutorial modules live in the RST directory, with the actual source in RST/source.

Code examples that will be presented within the modules (such as Processing or Python code) lives in the SourceCode directory.

Individual files are further labeled by functional type. Files related to AVs have their filename end in AV (such as insertionsortAV.js). Files related to proficiency exercises end in PRO. Files related to mini-slideshows or similar content that is included within a module end in CON. Khan Academy exercises that are multiple choice questions end in MC, and T/F questions end in TF. KA-based questions that are interactive (for example, where a user clicks on JSAV array elements to give an answer) end in PRO. The practice is to put individual KA questions in separate files, and often these are then aggregated to present to students as a battery of “summary” questions. Such aggregations end in Summ.

2.6 OpenDSA Coding Standards

Coding standards for OpenDSA are largely driven by validation tools. The requirements for CSS and JavaScript files are embedded in the validation tools and settings built into the OpenDSA Makefile. No code gets out of the

"Development" stage and into public use until it follows our rules for splitting into separate HTML/CSS/JavaScript files and passes the validation tools with zero warnings and errors.

HTML Pages Ideally, HTML pages that are part of OpenDSA should pass the W3 validation suite. An easy way to run this on your page is to install the Web Developer plugin (from http://chrispederick.com) for your browser. This is available for both Chrome and Firefox, and gives you icons on your toolbar that lets you run the validator on the current page. Unfortunately, we so far have not adopted a command-line tool for validation of HTML pages similar to what we are using for CSS and JavaScript.

We try to avoid JavaScript and CSS in the HTML pages (though we often tolerate a couple of lines of CSS for an AV that needs only minimal customization away from the standard lib/odsaStyle.css template). Our standard practice is to use `<script>` and `<link>` tags to call separate .js and .css files, respectively.

CSS Files We use `csslint` to validate css files. OpenDSA/Makefile contains our required csslint flags.

JavaScript We use `eslint` for validating JavaScript. OpenDSA/.eslintrc contains our official configuration file that defines the expected style. It is relatively strict. Developers should strive to eliminate all warnings (and of course, all errors).

JSON Files We use `jsonlint` to validate css files.

2.7 Tools

This section describes the various tools that are either required or might be particularly helpful for various aspects of OpenDSA development.

2.7.1 git

There are several versions of git for Windows We recommend the version found at http://msysgit.github.com/. This guide assumes that windows users are working through the Git Bash command window. When installing `msysgit`, you should install it at the level of `C:\`, rather than in `C:\Program Files`. Otherwise, you are likely to have troubles interacting with `GnuWin32` tools (for `make`). Aside from this, just accept the default configuration options.

2.7.2 make

GNU make for Windows can be found at http://gnuwin32.sourceforge.net/packages/make.htm.

2.7.3 Python

We are using Python 2.7 (NOT 3.x).

2.7.4 Python setuptools

Python setuptools is used for installing Sphinx. On Linux this might come preinstalled. If not, run the following using the appropriate package manager for your distribution (on Ubuntu, it is "apt-get"):

```
sudo <package_manager> install python-setuptools
```

On windows, see http://pypi.python.org/pypi/setuptools#files. You will need to include [PythonHome]/Scripts on your PATH system variable for both setuptools and sphinx. I had some trouble installing setuptools for the 64-bit version of Python 2.7.3 on Windows. When I tried to install setuptools, it wouldn't recognize that a Python installation was available. This is a known problem. You can either re-install the 32-bit version, or look on the internet for the proper registry work-around.

2.7.5 sphinx

For documentation, see http://sphinx.pocoo.org/contents.html.

With Python and setuptools installed, just type `easy_install -U Sphinx` at the command line.

2.7.6 Hieroglyph

Hieroglyph is only needed to compile course slides. You need to use version 0.5.5 (newer versions don't work). To install, just type `easy_install pip; pip install hieroglyph==0.5.5` at the command line.

2.7.7 nodejs

We don't use nodejs directly in our toolchain, but this is useful for installing several of the other tools. For installation instructions, see http://nodejs.org (and don't forget to check for the 64-bit version if that is the OS you are running).

2.7.8 eslint

Once you have nodejs installed, just do:

```
npm install -g eslint
```

2.7.9 csslint

Once you have nodejs installed, just do:

```
npm install -g csslint
```

Note: To be able to lint check either JavaScript or CSS, you need to put it in separate files from your HTML code.

2.7.10 jsonlint

Once you have nodejs installed, just do:

```
npm install -g jsonlint
```

2.7.11 uglifyjs

We use this for minimizing JavaScript code. To install on Windows:

```
npm install -g uglify-js
```

On Linux, you more likely will need to use the package manager. For example, on Ubuntu:

```
apt-get install uglifyjs
```

2.7.12 Some other things: requirements.txt

(This needs more documentation.) From the top level of the OpenDSA directory, do the following:

```
pip install -r requirements.txt
```

2.7.13 Notes for Windows

- You will need to be sure that Git, Python, and make are on your path. On Windows 7, you edit your path variable by right-clicking your Computer icon, clicking on “Advanced system settings” and then “Environment Variables”.
- If you have a 64-bit operating system, be aware that the various GNU tools will not work properly if they see “Program Files (x86)” on the system path variable. You might need to install these tools elsewhere, and/or reorder things on the path so that the GNU tool appears before anything referencing “Program Files (x86)”.
- We have had a lot of trouble getting the Git Bash shell to work properly when running GnuWin32 tools like “make”. One solution is to make sure that Git is not installed to a directory whose name has spaces in it (in particular, the standard “Program Files” directory that is the default). Instead, we find it most reliable to install Git directly into C:/.
- Beware if you have Cygwin installed on your Windows machine: There might be path conflicts between Cygwin on the one hand, and the Git Bash shell and the GNU tools on the other. If you insist on trying to use both on your system, you are on your own. Otherwise you have two reasonable options:
  - If you don’t use Cygwin much, then delete it entirely from your system.
  - Or stick completely with using Cygwin, by running Git and your other tools from within it instead of the Git command shell.

2.8 Web Programming Resources

Since we do so much webpage development and programming in JavaScript, newcomers will need good resources. One well-respected site is https://developer.mozilla.org/en/JavaScript. Beware of doing a search engine query and ending up at w3schools, which is not so well respected these days. If you are at Virginia Tech (or if your school supports this), a wonderful source of documentation is the Safari database (http://proquest.safaribooksonline.com/?uicode=viva), which contains a huge collection of technical books including the entire O’Reilly catalog.

2.9 Debugging

When you right-click a web page in Chrome (or Firefox when Firebug is installed), you get a popup menu whose bottom item is “Inspect Element”. This brings up the Chrome Developer Tools panel (in Chrome) or Firebug (in Firefox). This is especially helpful for inspecting the various DOM elements on your web page. A big help here is seeing the CSS styles in effect for any specified DOM element. For details on how to view and even edit on-the-fly your CSS settings in force (for example, to see what you should change), see https://developers.google.com/chrome-developer-tools/docs/elements-styles.

While Chrome has built-in developer tools (and a lite version of Firebug), we highly recommend using the full version of Firebug, available for Firefox, for JavaScript debugging. More information about Firebug’s features can be found here: https://getfirebug.com/.

The following are highlights for some debugger features and how they can be used.

- Console - an interactive JavaScript console which allows:
  - Print statments for debugging and error logging.
  - Testing JavaScript statements (including access to variables and functions defined on the current page).
  - Viewing network requests - GET and POST messages appear in the console allowing the user to see what data was sent and the server's response.
- Inspect - allows the user to select an element on a page, view the HTML for it and modify the element's CSS in real time (helpful for rapid GUI prototyping).
- Debugger - a full featured JavaScript debugger (useful for debugging or simply following code execution).

2.10 Setting up a Testing Environment

To compile your own books for testing purposes requires rather a lot of infrastruture. It also involves running multiple servers: at least one for the LTI provider and one for the OpenDSA scoring server. To make this relatively easy for most developers, we have created a package to deliver a complete "OpenDSA in a box" on a virtual machine. Complete instructions can be found at: https://github.com/OpenDSA/OpenDSA-DevStack.

CHAPTER 3

Instructor's Tools

3.1 OpenDSA and the LMS

OpenDSA now works in conjunction with a Learning Managment System (LMS). Initially, only Canvas is supported, but we hope to support more systems in the near future. All tasks related tc class roll management, creating assignments from exercises, due dates, and tracking scores and progress are done using the standard tools provided by the LMS.

CHAPTER 4

Compiling Book Instances: OpenDSA Configuration

4.1 Overview

A given OpenDSA eTextbook is called a “book instance”. The contents of a book instance is defined by a configuration file, the detailed syntax for which is defined in this section. In practice, it is easiest to start by copying an existing configuration file, and then changing it to fit your needs. Configuration files are JSON files, normally stored in OpenDSA/config. From the top level of an OpenDSA repository, you can compile a book instance (given the existance of a configuration file named `config/foo.json`) by issuing this command:

```
python tools/configure.py config/foo.json
```

Separate from book instances is the concept of a “course instance”. Depending on the course software (typically an LMS such as Canvas or Moodle) that you are using, you will likely need to compile a special instance of a given OpenDSA book instance for each course that intends to use it. OpenDSA course instances are managed by a separate configuration file. The convention is that if the book configuration file is `foo.json`, then the corresponding course configuration file(s) will be in in `fooXX_LMSconf.json` where XX allows you to distinguish between the various courses that use the book.

Details about how to set up a course is in *Configuring Courses* below.

4.2 Motivation for the Configuration System

- Allows content to be environment-independent
  - Configuration file contains all environment-dependent settings such as paths and target URLs
  - Example: If developers want to point their front-end code at different backend systems, they simply make the change in their own config file. They can share all OpenDSA content, but when they build the book, it will be built using their personalized settings
- Allows easy replication
  - Collects all settings and preferences required to configure an instance of OpenDSA in a single, portable file that can be easily shared among instructors.
  - Once a configuration has been created, instructors can make identical copies without going through the configuration process
- Allows fine grain control
  - Existing configuration files provide sensible defaults, but allow instructors to control aspects such as how many points a specific exercise is worth or whether it is required for module proficiency

- Configuration files will eventually be generated by a GUI front-end, but for now we create them by hand (tedious, and the motivation for a GUI tool is rapidly moving to the front burner).

4.3 Branches

We are temporarily supporting two versions of OpenDSA, as represented by the branches "master" and "NewKA". The "master" branch currently is intended for generating books as used in Canvas, while "NewKA" supports "old style" book instances directly accessible via HTML files. Only the master branch uses the server configuration files described at the end of this section. There are a few fields of the configuration process that appear in NewKA book configuration files, that are moved to the server configuration file in the master branch.

4.4 Configuration Process

A comment at the beginning of configure.py gives some information information about how the script works. Generally, those who want to compile a book instance do not need to worry about those details.

Note: The OpenDSA root directory must be web-accessible as there are many supplemental directories and files (AV/, Exercises/, lib/, SourceCode/, etc) which must be referenced. These files are identical for all books and not copying them reduces the footprint of each book.

4.4.1 Module and Exercise Removal

- Only the modules listed in the configuration file will be included. To remove a module from the book, simply remove the module object from the configuration file.
- To remove a section from a module, set the "showsection" attribute to `false`. Exercises are normally the sole contents of some section. Note that Exercises that do not appear in the configuration file will still be included in the book using the default configuration options. During compilation, a list will be printed of any exercises which were encountered in the modules but not present in the configuration file.

4.4.2 Book Name

The name of the configuration file will often be used in some way by the LMS that manages the course. For example, the name of the configuration file might be the Course ID in the LMS.

4.5 Future Features

- GUI editor/interface for editing configuration files.
- Merge the "master" and "NewKA" branches, with support for both Canvas-style and "HTML page" book instances, controlled by command-line parameters on the configuration script.

4.6 Format

OpenDSA configuration files are stored using JSON. Here are the field definitions. All are required unless otherwise specified. **Note**: The fields "title", "exercise_server", "logging_server", and "score_server" appear in the book configuration file only for the NewKA branch. These fields are moved to the server configuration file in the master branch.

- **title** - The title of the OpenDSA textbook.
- **build_dir** - (optional) The directory where the configured book directory will be created, defaults to 'Books' if omitted
  - A new directory, named after the configuration file, will be created at this location and serve as the output directory for the configuration process. Files required to compile the book will be copied / written to the output directory, including modified version of the source RST files
    - Example: Assume "build_dir": "Books" and the name of the configuration file used is "CS3114.json", the output directory would be "~OpenDSA/Books/CS3114/"
  - The compiled textbook will appear in `[build_dir]/[book name]/html`
  - This directory must be web accessible
- **code_dir** (optional) - Specifies a directory that contains another directory as specified by `code_lang` (see below). Defaults to `SourceCode` if omitted.
  - Ex: Using `"code_dir": "SourceCode/"`, and assuming that the defined language directory is `Python` then the configuration process would look for example Python source code in `~OpenDSA/SourceCode/Python`.
- **code_lang** - A dictionary where each key is the name of a programming language (supported by Pygments and matching a directory name in `code_dir`), and each value is a dictionary of language options. Language options are:
  - `ext` for a list of file extensions.
  - `label` for the text that will be displayed at the header of the code snippet tab.
  - `lang` for the name of the programming language (supported by Pygments).

  The order in which the languages and extensions are provided determines their order of precedence in cases where only one display code is to be selected.
  - Example:

    ```
    "code_lang": {
       "C++": {"ext": ["cpp","h"],"label":"C++","lang":"C++"}
       "Java": {"ext":["java"], "label":"Java", "lang":"java"},
       "Processing": {"ext":["pde"], "label":"Processing","lang":"java"}
    }
    ```
  - In this example, assuming that `code_dir` is `SourceCode/`, the system would search for `.cpp` files, followed by `.h` files in `~OpenDSA/SourceCode/C++/`, then `.java` files in `~OpenDSA/SourceCode/Java/`, and finally `.pde` files in `~OpenDSA/SourceCode/Processing/`.
  - There is not actually a need for the "code" language files to be a real programming language. For example, it is plausible to use a "programming language" called `Pseudo` with file suffix `.txt`. However, Pygments might not do well with colorizing the result.
- **tabbed_codeinc** (optional) - A boolean that controls whether or not code is displayed in a tabbed interface. If true, it will display the specified code in each of the languages specified in `code_lang` (if the code exists) in a tabbed container. If false, it will display the code in a single language (the first language for which the code exists with the order of precedence determined by the order specified in `code_lang`). Defaults to `true` if omitted.

- **lang** (optional) - Specifies the native language of the book using the official ISO 639-1 or 639-2 standard abbreviation, defaults to `en` if omitted. This is used to control where RST source files are located, within `~OpenDSA/RST/[lang]`. Any RST files not found in the indicated subdirectory will then be located in `~OpenDSA/RST/en`.
- **module_origin** - The protocol and domain where the module files are hosted
  - Used by embedded exercises as the target of HTML5 post messages which send information to the parent (module) page
  - Ex: "module_origin": "http://algoviz.org",
- **av_root_dir** - (optional) Allows the user to change the default location where the `AV/` directory can be found. Defaults to `~OpenDSA/` if omitted
  - This can point to another location on the same machine that hosts the module files (as long as it is web-accessible), or it can point to a remote location (this feature not supported yet).
  - **Note**: This should not point to the AV/ directory itself, but instead should be the directory containing the AV/ directory (to avoid breaking the relative paths in the RST files).
  - If this attribute references a remote location, 'av_origin' must be present and the value must be a prefix of the remote location.
  - Ex: "av_root_dir": "/home/algoviz/OpenDSA/test/",
  - Ex: "av_root_dir": "http://algoviz.org/OpenDSA/", // This directory contains an AV/ directory
- **av_origin** - (normally optional, but required if **av_root_dir** is defined) The protocol and domain where the AV files are hosted, defaults to match `module_origin` if omitted.
  - Used on module pages to allow HTML5 post messages from this origin, allows embedded AVs to communicate with the parent module page.
  - Ex: "av_origin": "http://algoviz.org",
- **glob_mod_options** - (optional) An object containing options to be applied to every module in the book. Can be overridden by module-specific options.
- **glob_exer_options** - (optional) An object containing options to be applied to every exercise in the book. Can be used to control the behavior of the exercise. Can be overridden by exercise-specific options.
- **exercises_root_dir** - (optional) Allows the user to change the default location where the `Exercises/` directory will be found. Defaults to `~OpenDSA/` if omitted.
  - This can point to another location on the same machine that hosts the module files (as long as it is web-accessible) or it can point to a remote location (not fully supported yet).
  - **Note**: This should not point to the `Exercises/` directory itself, but rather the directory containing the `Exercises/` directory (to avoid breaking the relative paths in the RST files)
  - If this attribute references a remote location, `exercise_origin` must be present and the value must be a prefix of the remote location
  - If this attribute is not present, `~OpenDSA/` will be used as the default.
  - Ex: "exercises_root_dir": "/home/algoviz/OpenDSA/test/",
  - Ex: "exercises_root_dir": "http://algoviz.org/OpenDSA/", // This directory contains an Exercises/ directory
- **exercise_origin** - (optional, unless **exercises_root_dir** is defined) The protocol and domain where the Exercises files are hosted, defaults to match `module_origin` if omitted.

  - Used on module pages to allow HTML5 post messages from this origin, allows embedded exercises to communicate with the parent module page.
  - Ex: "exercise_origin": "http://algoviz.org",
- **exercise_server** - (optional) The protocol and domain (and port number, if different than the protocol default) of the exercise server that provides verification for the programming exercises. Defaults to an empty string (exercise server disabled) if omitted.
  - Trailing '/' is optional
  - Ex: "exercise_server": "https://opendsa.cs.vt.edu/",
- **logging_server** - (optional) The protocol and domain (and port number, if different than the protocol default) of the logging server that supports interaction data collection. Defaults to an empty string (logging server disabled) if omitted.
  - Trailing '/' is optional
  - Ex: "logging_server": "https://opendsa.cs.vt.edu/",
- **score_server** - (optional) The protocol and domain (and port number, if different than the protocol default) of the score server that supports centralized user score collection. Defaults to an empty string (score server disabled) if omitted.
  - Trailing '/' is optional
  - Ex: "score_server": "https://opendsa.cs.vt.edu/",
- **build_JSAV** - (optional) A boolean controlling whether or not the JSAV library should be rebuilt whenever the book is compiled. Defaults to `false` if omitted.
  - This value should normally set to `false` for development.
  - Instructors may wish to set this to true for production environments when configuration is run infrequently and JSAV is likely to have changed since the last time configuration occurred.
- **build_cmap** - (optional) A boolean controlling wether or not the glossary terms concept map should be diplayed. Defaults to `false`.
- **allow_anonymous_credit** - (optional) A boolean controlling whether credit for exercises completed anonymously (without logging in) will be transferred to the next user to log in. Defaults to `true` if omitted. **Note:** Obsolete in the context of LMS support for scoring, since the LMS will require login for access to the OpenDSA content.
- **req_full_ss** - (optional) A boolean controlling whether students must view every step of a slideshow in order to obtain credit. Defaults to `true` if omitted.
- **start_chap_num** - (optional) Specifies at which number to start numbering chapters. Defaults to 0 if omitted.
- **suppress_todo** - (optional) A boolean controlling whether or not TODO directives are removed from the RST source files. Defaults to `false` if omitted.
  - **Note**: When changing from `false` to `true`, you must run `make clean` or otherwise remove previously compiled book files so as to completely remove any references to `ToDo`.
- **tag** - (optional) A string containing a semi-colon delimited list of tags. This directs Sphinx to include material from RST `only` directives with the matching tag(s). This is useful for relatively fine-grain control over whether material will be included in a book instance. For example, if you want to have multiple paragraphs each with a programming language-dependent discussion, with only the appropriate paragraph for the language being used for this book instance actually appearing to the reader. Any material within an `only` block that does **not** have a matching `tag` in the config file will be left out.

- **assumes** - (optional) A string containing a semi-colon delimited list of topics that the book assumes students are familiar with. This allows for control over warnings about missing prerequisite modules during the build process.
- **chapters** - A hierarchy of chapters, modules, and sections. This makes up the vast majority of most configuration files.
  - Each key in "chapters" represents a chapter name. A module object is one whose key matches the name of an RST file in the `~OpenDSA/RST/[lang]/` directory, and which contains the key "sections".
  - **hidden** - This is an optional field to signal the preprocessor to not display the content of the chapter in the TOC. The configuration script will add the new directive `odsatoctree`. The flagged chapter entries in the TOC will be of class `hide-from-toc`, and will be removed by a CSS rule in odsaMOD.css file.
  - **Modules**
    - The key relating to each module object must correspond to a path to an RST file found in ~OpenDSA/RST/[lang]/.
    - **long_name** - A long form, human-readable name used to identify the module.
    - **dispModComp** - (optional) A flag that, if set to "true", will force the "Module Complete" message to appear even if the module contains no required exercises. If set to "false", the "Module Complete" message will not appear, even if the module DOES contain required exercises.
    - **mod_options** - (optional) overrides `glob_mod_options`, which allows modules to be configured independently from one another. Options that should be stored in `JSAV_OPTIONS` should be prepended with `JOP-` and options that should be stored in `JSAV_EXERCISE_OPTIONS` should be prepended with `JXOP-`. (This can be used to override the defaults set in `odsaUtils.js`). All other options will be made directly available to modules in the form of a parameters object created automatically by the client-side framework (specifically, when `parseURLParams()` is called in `odsaUtils.js`).
    - **sections** - A collection of section objects that define the sections that make up a module. The `sections` object should contain keys that match the titles of the corresponding sections in the RST file. Some modules contain no sections, in which case this field should be included with an empty list.
      - To remove the section completely, provide the field `showsection` and set it to `false`.
      - All options provided within a section object (with the exception of `remove`) are appended to the directive, please see the *Extensions* section for a list of supported arguments.
      - A section may contain a single exercise descriptor, as follows.
      - **exer_options** - (optional) An object containing exercise-specific configuration options for JSAV. Can be used to override the options set using `glob_exer_options`. Options that should be stored in `JSAV_OPTIONS` should be prepended with `JOP-` and options that should be stored in `JSAV_EXERCISE_OPTIONS` should be prepended with `JXOP-`. (This allows overriding the defaults set in `odsaUtils.js`.) All other options will be made directly available to exercises in the form of a parameters object created automatically by the client-side framework (specifically when `parseURLParams()` is called in `odsaUtils.js`).
      - **long_name** - (optional) A long form, human-readable name used to identify the exercise. Defaults to short exercise name if omitted.
      - **points** - (optional) The number of points the exercise is worth. Defaults to `0` if omitted.
      - **required** - (optional) Whether the exercise is required for module proficiency. Defaults to `false` if omitted.
      - **threshold** - (optional) The percentage that a user needs to score on the exercise to obtain proficiency. Defaults to 100% (1 on a 0-1 scale) if omitted.

        · JSAV-based diagrams do not need to be listed

    * **codeinclude** (optional) - An object that maps the path from a codeinclude to a specific language that should be used for that code. This allows control of individual code snippets, overriding the `code_lang` field.

        · Ex: `"codeinclude": {"Sorting/Mergesort": "C++"}` would set C++ as the language for the codeinclude "Sorting/Mergesort" within the current module.

4.7 Configuring Exercises

The most important concern when configuring proficiency exercises is the scoring option to be used. JSAV-based proficiency exercises have a number of possible grading methods:

- `atend`: Scores are only shown at the end of the exercise.
- `continuous:undo`: Mistakes are undone, the student will lose that point but have to repeat the step.
- `continuous:fix`: On a mistake, the step is corrected, the student loses that point, and then is ready to attempt the next step. This mode requires that the exercise have the capability to fix the step. If it does not, this grading mode will default to `continuous:undo`.

All proficiency exercises can be controlled through URL parameters. What the configuration file actualy does by setting `exer_options` is specify what should be in the URL parameters that are sent to the exercise by the OpenDSA module page. Here is an example for configuring an exercise:

```
"shellsortPRO": {
  "long_name": "Shellsort Proficiency Exercise",
  "required": true,
  "points": 2.0,
  "threshold": 0.9,
  "exer_options": {
    "JXOP-feedback": "continuous",
    "JXOP-fixmode": "fix"
  }
},
```

This configuration will affect the configuration of an entity called `shellsortPRO` (presumeably defined by an `..avembed` directive in the corresponding OpenDSA module). It is scored (as indicated by setting the `required` field to `true`), and is worth 2.0 points of credit once the user reaches "proficiency". To reach "proficiency" requires correctly achieving 90% of the possible steps on some attempt at the exercise (as defined by `threshold`). The exercise is instructed to use the `continuous:fix` mode of scoring.

In addition to the standard `JXOP-feedback` and `JXOP-fixmode` parameters, a given AV or exercise might have ad hoc parameter settings that it can accept via URL parameter. Examples might be algorithm variations or initial data input values. Those would have to be defined by the exercise itself. These (along with the standard grading options) can also have defaults defined in the `.json` file associated with the AV or exercise, which might help to document the available options. Any such ad hoc parameter defaults can be over-ridden in the `exer_options` setting in the configuration file.

4.8 Configuring Courses

4.8.1 Rationale

Separate from book configuration files (which define the contents of a book, scoring information, and configurations for various exercise), a given book instance will typically be accessed in the context of a particular LMS, which will require various permissions in order to operate correctly. The compilation process separates the compilation of book files from the interactions needed to set up the book's use at a specific instance of the LMS. Book instances are in fact compiled to the specification necessary for that specific LMS to access it, meaning that book instances cannot be shared across LMS's, or by different instances of the same LMS (say, two instances of Canvas), or even by two course instances on the same installation of a given LMS. The reason is that the internal cross links between the various parts of the book instance are often defined in the context of a specific course instance within the LMS.

A specific course instance on a specific LMS installation is defined by a course configuration file. By convention, the file name will end with `_LMSconf.json`. A template for course configuration can be found here.

Since course configuration files routinely store sensitive information such as account passwords and access keys, they are not stored in the OpenDSA repository. This documentation along with the template file should provide enough information for you to successfully define the contents of a configuration file.

A set of `make` targets are available in the OpenDSA Makefile. From the top level of an OpenDSA repository, you can compile the HTML files for a book instance by typing

```
make <courseinstance>
```

So, if there existed a book with a configuration file named `config/foo.json`, you would type

```
make foo
```

This much (locally creating the HTML files) uses just the book configuration file.

If you want to bind a book instance to a particular course instance on a given LMS, that requires both compiling the book and pushing information about it to the LMS. Pushing information to the LMS is where the course configuration file comes into play. **After** you set up the proper course configuration file in `config/foo_LMSconf.json`, you can type:

```
make foo opts="-c True"
```

to push the necessary book data to your LMS. Alternatively, there may already be a Makefile target named `fooLMS` that has this same effect by typing

```
make fooLMS
```

4.8.2 Format

To understand the following description of configuration file data fields, it helps to understand that running a "course" using OpenDSA requires communication between several entities, including:

- An LTI tool provider. This is the site that hosts the book, which is probably where the book is being compiled.
- An LMS. The LMS has to grant access to the LTI provider in order for it to send scores and define the modules.
- The OpenDSA scoring, logging, and programming exercise server(s). Communications with these are required in order to handle crucial aspects of exercise scoring.

Here are the fields in the configuration file.

- **title** - The title for the course intance.
- **odsa_username** - A viable user account on the course instance (OpenDSA) scoring server.

- **odsa_password** - The corresponding password on the course instance (OpenDSA) scoring server.
- **target_LMS** - LMS name. We currently support 'canvas'. Other LMSs like moodle and Desire2Learn will be supported in the future.
- **LMS_url** - The URL for the LMS.
- **course_code** - The name used at the LMS to identify the course. In Canvas, this is the identfier given when creating the course.
- **access_token** - This is normally issued by the LMS to allow an LTI tool provider to communicate with it. In Canvas, go to your account-level settings. Near the bottom of the page you should see a big blue button that reads "New Access Token". Click this, then copy the string that is generated, and paste it into this field in the configuration file. If you (the creator of the config file and the one who compiles the book) are not the course instructor (with access to the LMS), then the course instructor will need to provide this access token.
- **LTI_consumer_key** - The key required by the LTI tool provider.
- **LTI_secret** - Effectively the password for the LTI tool provider.
- **module_origin** - The protocol and domain where the module files are hosted
  - Used by embedded exercises as the target of HTML5 post messages which send information to the parent (module) page
  - Ex: "module_origin": "http://algoviz.org",
- **exercise_server** - The protocol and domain (and port number, if different than the protocol default) of the exercise server that provides verification for the programming exercises. Defaults to an empty string (exercise server disabled) if omitted.
  - Trailing '/' is optional
  - Ex: "exercise_server": "https://opendsa.cs.vt.edu/",
- **logging_server** - The protocol and domain (and port number, if different than the protocol default) of the logging server that supports interaction data collection. Defaults to an empty string (logging server disabled) if omitted.
  - Trailing '/' is optional
  - Ex: "logging_server": "https://opendsa.cs.vt.edu/",
- **score_server** - The protocol and domain (and port number, if different than the protocol default) of the score server that supports centralized user score collection. Defaults to an empty string (score server disabled) if omitted.
  - Trailing '/' is optional
  - Ex: "score_server": "https://opendsa.cs.vt.edu/",

CHAPTER 5

Module Authoring

OpenDSA modules are authored using reStructuredText (also known as ReST). The source files for these modules (which can be found in `OpenDSA/RST/source`) are compiled to HTML (or, in theory, other formats – but we do not support that) by Sphinx. To create a "book", you must invoke `OpenDSA/tools/configure.py`, being sure to specify a configuration file (sample of which can be found in `OpenDSA/config`. Sample books can be compiled using `OpenDSA/Makefile`. A sample command (run from the OpenDSA toplevel directory) looks like: `python tools/configure.py config/OpenDSA.json`.

A number of special directives have been created, which are documented in *OpenDSA ReST Extensions*.

Documentation for writing OpenDSA exercises using the Khan Academy infrastructure is in *Using OpenDSA with Khan Academy infrastructure*.

Documentation for writing pure JSAV-based proficiency exercises is in *Using OpenDSA with Khan Academy infrastructure*.

The best way to get a sense for how things work is to look at some existing modules.

5.1 Module Structure

Each module is basically a single ReST file. The first thing that will normally appear is the `avmetadata` directive block. Among other things, this will define the module within the prerequisite structure, which is important when generating a full textbook.

A big reason why we chose to use ReStructuredText for authoring is its ability to pass raw HTML through, allowing us to embed dynamic content (i.e., JavaScript) into our HTML pages while still having the advantage of a markup language for authoring. However, we don't ever want to actually use the `raw` directive in our modules if we can avoid it. At this point use of `raw` should never be needed, as we have a number of directives to use instead: `avembed`, `inlineav`, `odsalink`, and `odsascript`.

Most exercises and visualizations are embedded into the module from elsewhere using the `avembed` directive, but small slideshows and dynamically generated diagrams can be included directly using the `inlineav` directive. Any CSS or JS code that is unique to a specific module page should be maintained in separate files and included using `odsalink` and `odsascript` directives, respectively.

Defining which modules will be used in a given book, which exercises are included for credit, and various other aspects of module use are defined with the *Configuration system*.

5.2 Math and Symbol Escapes

All equations within a module are created using LaTeX syntax embedded in a `:math:` inline directive. This will be converted to appropriate math layout in the resulting HTML file. Note that due to various interactions between reStructuredText and MathJax (which does the LaTeX conversion within an HTML page), you have to use a double backslash for escaping the dollar sign symbol, such as:

```
This costs \\$5.00.
```

5.3 Math and Code

One of the hardest things when writing modules is making sure that all of the variables and expressions are marked up correctly. In nearly all cases, any variable is either "mathy" or "codey". Mathy variables and expressions should use LaTeX markup embedded in a `:math:` directive. "Codey" variables and expressions should be marked up as:

```
``my codey text``
```

All variables (and expressions) should always get their appropriate typeface. Avoid using physical markup such as italics or bold for such things, we prefer to use logical markup (that is, math markup or code markup). Sometimes it can be difficult to decide which is appropriate. For example, you might have a function with a variable `n` for the array size. When it comes time to discuss the analysis of the function, it is probably going to be done in terms of n, a quantity that expresses the array size (as opposed to the function variable `n`). It can be a subtle point whether the variable or the quantity is intended. Having to typeset it (and so make a conscious decision) helps you to think through what you are trying to convey.

5.4 Code Snippet Support

OpenDSA and JSAV provide an extensive framework for integrating code snippets into your modules and visualizations. JSAV provides support through the Pseudocode API for displaying and manipulating your code snippets within an AV. See the JSAV manual for details. Within a module, code snippets are meant to be embedded from a sourcecode file using the `codeinclude` directive. The default coding language(s) used by a textbook instance is controlled by the `code_lang` setting in the corresponding OpenDSA *configuration* file.

The OpenDSA framework and configuration support makes it as easy as possible to be able to compile book instances with code snippets from your desired programming language(s), assuming that the code snippets have been provided by a content developer. The most important principle for managing code snippets is that they should be taken from working programs that can properly support testing of the code that you include into your modules.

All such sourcecode should appear in the `SourceCode` directory within OpenDSA, with each coding language having its own subdirectory. A given AV can have an associated `.json` file that defines the configuration for alternate coding languages, including such things as the filename.

Note that in the `.json` file, a given section of the `code` block should match the subdirectory name within the `SourceCode` directory for that language.

5.5 Creating Course Notes

OpenDSA uses hieroglyph a Sphinx extension to build HTML slides.

The course notes infrastructures is similar to eTextBook creation, and uses `OpenDSA/Makefile`. The only difference is the `s` option for slides when calling the configuration, for example `python tools/configure.py s config/OpenDSA.json`.

5.6 Internationalization Support

OpenDSA supports a sophisticated internationalization framework that attempts to make it as easy as possible to support compiling textbook instances in various (natural) languages. The configuration system allows a book compiler to specify the language of choice, and the system will take module versions in the target language whenever available (the fallback language is English).

As a module author, your `.rst` files will always appear in a subdirectory of the `RST` directory coded to the language that you are writing for. Like every other aspect of internationalization, we define these subdirectories using the two-letter ISO 639-1 language codes. Thus, all English-language RST files appear in the `RST/en` directory.

CHAPTER 6

House Style Rules

The following guidelines should be observed when writing content for the OpenDSA project.

6.1 Content Types

OpenDSA content can generally be divided into the following types:

- modules: written in ReST and whose source resides in the OpenDSA/RST/source directory,
- exercises: implemented using the Khan Academy Infrastructure and residing in the OpenDSA/Exercises directory,
- mini-slideshows and diagrams: directly embedded into modules using the `inlineav` directive and whose source resides in the OpenDSA/AV directory, and
- visualizations and activities: embedded as iframes using the `avembed` directive, and residing in the OpenDSA/AV directory.

6.2 Algorithm Visualizations and slideshows

New content always should begin development within the OpenDSA/AV/Development directory. Once the implementation is completed and validated, then it will be moved by the project managers to a topic sub-directory.

Directly embedded content (slideshows and diagrams) should have no width set so that it will naturally expand to the page width as appropriate.

Nothing should require a width greater than 850px. This keeps all visual elements within the minimum page size enforced by the Haiku template.

The typical classroom projector provides a vertical screen resolution of 800 pixels, and that fits the limits of many laptops and tablets. Given the requirements for browser toolbars, etc., the most that you can count on for the vertical space shown for an actual browser page is about 650 pixels, so that it the absolute maximum height for any given visual element such as an AV or an exercise. It is better to stay under 600 pixels if possible, so that the show/hide button is also visible (this provides important feedback when working an exercise).

HTML, CSS, and JavaScript should all be kept in separate files. Each should pass its respective validation tools without complaint. See the developer's "Getting Started" guide for details on using the validation tools.

CHAPTER 7

Notes for AV and Exercise Developers

7.1 Configuration: Code lines and Internationalization

Any JSAV-based exercise or AV (either standalone or an in-lined slideshow) can be associated with a configuration file. This is a `.json` file whose default name is the same as the name of the container for an inlined slideshow, or the same as the standalone AV or exercise. Configuration files support sections for defining all strings (used for internationalziation support), mapping logical names to code lines (to support alternate programming language examples in JSAV code objects), and setting defaults for configuration parameters.

7.1.1 File Format

The file format needs to be documented. In the meantime, a good example is `AV/Sorting/insertionsortAV.json`.

7.1.2 Use Case: Standalone AV or Proficiency Exercise

Standalone AVs and proficiency exercises are embedded into the HTML page using an iframe. The size of the iframe is taken from the size of a `div` element with classname `container`.

Given a standalone AV with HTML file `foo.html` that contains a `div` with classname `avcontainer` and JavaScript file `foo.js`, the configuration file would normally be named `foo.json`. After creating a configuration object, the string interpreter and code interpreters will typically be created as follows:

```
av = new JSAV($(".avcontainer"));
// Load the config object with interpreter and code created by odsaUtils.js
var config = ODSA.UTILS.loadConfig(),
    interpret = config.interpreter,       // get the interpreter
    code = config.code;                   // get the code object
var pseudo = av.code(code);
```

7.1.3 Use Case: Inline Slideshow

The RST file will reference a given slideshow with a specified `div` name as:

```
.. inlineav:: fooCON ss
```

The corresponding JavaScript file that implements the slideshow will create the JSAV, config, interpreter, and code objects with code like:

```
var av_name = "fooCON";
// Load the config object with interpreter and code created by odsaUtils.js
var config = ODSA.UTILS.loadConfig({"av_name": av_name}),
    interpret = config.interpreter,       // get the interpreter
    code = config.code;                   // get the code object
var av = new JSAV(av_name);
var pseudo = av.code(code);
```

7.1.4 Use Case: No pseudocode object

Creating the `code` object is not necessary if no pseudocode object is used in the visualization. A shortened version is therefore as follows:

```
var av_name = "fooCON";
var interpret = ODSA.UTILS.loadConfig({"av_name": av_name}).interpreter;
var av = new JSAV(av_name);
```

7.1.5 Use Case: Shared JSON file

Occasionally you might have multiple slideshows that share code or strings, such that it makes sense for them to share a configuration file. In this example, consider a JavaScript file `fooCON.js` that contains the code for one or more slideshows, with shared configuration file `fooConfCON.json`. The associated RST file would contain a line for each slideshow, such as:

```
.. inlineav:: fooS1CON ss
```

Each slideshow implemented in `fooCON.js` would contain code similar to:

```
var av_name = "fooS1CON";
// Load the config object with interpreter and code created by odsaUtils.js
var config = ODSA.UTILS.loadConfig(
               {"av_name": av_name, "json_path": "AV/Topic/fooConfCON.json"}),
   interpret = config.interpreter,       // get the interpreter
   code = config.code;                   // get the code object
var av = new JSAV(av_name);
var pseudo = av.code(code);
```

7.1.6 Using the configuration

After creating the interpreter and code objects, they can be used to replace text strings and line numbers as follows:

```
// If the .json file has definitions for av_c2 and av_c3 in various languages:
av.umsg(interpret("av_c2"));
av.label(interpret("av_c3", {top: 10, left: 10}).show();
// If the .json file has definitions for codelines with the tag ``loop``:
pseudo.setCurrentLine("loop");
```

7.2 URL Parameters

The client-side framework provides functionality to support easy processing of URL paramters in any stand-alone AV or exercise. See `parseURLParams()` in the *Client-side Development* section.

Some URL parameters are considered a standard part of the system and have built-in support. These include the parameters to control the natural language, the programming language (see *Internationalization Support*), and the exercise grading options. AV/exercise developers can also implement support for their own ad hoc URL parameters.

When an AV or exercise is embedded in an OpenDSA module via the `avembed` directive, the URL parameters are controlled by the *configuration process*. However, if a third party wishes to call the stand-alone AV or exercise directly (perhaps someone will want to embed calls within their own HTML pages), URL parameters are invoked as follows.

> <URL>?JXOP-code=java

This one directs the AV to display Java code.

> <URL>?JOP-lang=fi

This one directs the AV to use Finnish for its text.

Proficiency exercises typically support various grading modes. These are documented in the *Configuration* section. The typical options are as follows:

```
<URL>?JXOP-feedback=atend
<URL>?JXOP-feedback=continuous&JXOP-fixmode=undo
<URL>?JXOP-feedback=continuous&JXOP-fixmode=fix
```

7.3 Equations

Within `jsav.umsg()` text, all math should be done using LaTeX format enclosed within `$...$` (for inline expressions) or `$$ ... $$` (for display equations). MathJax will automatically recognize the dollar sign markup, and it will automatically do the conversion from LaTeX format to HTML. The only peculiarity that you should need to worry about is that backslashes must be escaped by using two backslashes. So a typical math markup within an AV or slideshow might look like:

```
jsav.umsg("This takes $\\Theta(n)$ time.");
```

7.4 CSS

Anything related to visual element style that is static should be defined in a CSS file. For example, if a JSAV array is placed at a specific location that never changes, then this location should be defined within a CSS file for your AV or slideshow.

While the client-side framework should automatically resize the AVs iFrame, developers should set the default height and width of the AV to accommodate the maximum size of the AV (such as an optional code block). If the automatic resizing should fail, the exercise should still be useable even if it doesn't look as nice.

Some styling aspects are dynamic. For example, over the course of a visualization, nodes in a tree might need to change color to emphasize the action being visualized. Looking at the JSAV manual, you will notice that most visual elements can be styled with a `.css()` method on the element. But in nearly all cases, we wish to avoid using that method. We prefer to use the `.addClass()` and `.removeClass()` methods to control dynamic element styling whenever possible. These methods will dynamically assign or remove a CSS class to the element in the DOM. You can define any necessary new class in your AV's CSS file. But before doing so, you should first check to see if a suitable class already exists in the OpenDSA style file at `lib/odsaStyle.css`. Given that we have developed a lot of visualizations already, the odds are pretty high that whatever visual styling you want to do is semantically equivalent to something that we already support. If so, you should be using the same style definition. For example, if you have a cell in an array or a node in a tree that your AV is currently acting on, then you probably want to indicate this by styling it using `mynode.addClass("processing")` for a tree node object named `mynode`, or using `myarray.(index, "processing")` for array position `index` in JSAV array `myarray`.

7.5 "Stand-alone" vs. "Inline" AVs and Exercises

Structurally, there are two ways that we include AVs and exerices into a module. First is the "stand-alone" artifact, which has its own HTML pages. In principle, this might be anything with its own URL, though in practice we usually only include our own materials. This is done using the `avembed` directive (see *avembed*). When converted to HTML, the mechanism used is a standard `iframe` tag to include the artifact.

"Inline" AVs are usually either a JSAV diagram or a JSAV slideshow (a diagram is just a "slideshow" with no slide controls at the top). These are included using the `inlineav` directive (see *inlineav*). The `avID` is the container name for the AV. Of course, the final HTML page has to get access to the relevent JavaScript and CSS files. This is done by putting at the bottom of the .rst file an `odsascript` directive giving the path and name of the Javascript file (see *odsascript*). If a CSS file is used, then you put near the top of the .rst file (right after the `avmetadata` block) an `odsalink` directive giving the path and name of the CSS file (see *odsalink*). Our naming convention is that all inlineavs use container names that end in `CON`, and that the .js and .css files use the container name. Further, our convention is that each individual slideshow or diagram be in its own JavaScript file (though this is convention is violated on occasion if there are a lot of very short slideshow files in a given page).

The `odsascript` and `odsalink` directives do nothing more than map down to `<script></script>` and `<link></link>` tags, respectively, in the final HTML pages. Their purpose is merely to keep module authors from needing to use raw HTML code in an RST file.

When you embed multiple slideshows on the page (with `inlineav`), they will naturally share the same namespace, both for code and for CSS.

For code, this is not generally an issue, because it is our standard procedure to wrap all of our code in an "anonymous function", and then reference the key identifier (the container div) by name. This is why you will always see (in any of our code that has been cleaned to our internal spec, which should be everything except perhaps code in the Development directory) something like the following:

```
$(document).ready(function () {
  var av_name = "insertionsortS1CON";
  ...
  var av = new JSAV(av_name);
  ...
});
```

This does the following:

- document.ready makes it wait until everything is loaded
- It is all wrapped in a function, so that its namespace will not conflict with other slideshows. That way, for example, the global variables for one slideshow (like `av` in this example) are separate from the other slideshows. (This actually causes a problem if you want to include functions from other .js files. See *Encapsulation*.)
- Use of the container name (such as in the JSAV call) is why THIS code gets executed on THIS container instead of the OTHER .js files that you loaded on the page.

Each `inlineav` might need to set some CSS styling with the same name as other slideshows will use. You handle this by "qualifying" the relevant variable to the name of the div that contains it. Look for example at `AV/Binary/BSTCON.css` to see examples. Notice lines that look like:

```
#avnameCON .jsav.jsavtreenode {
  ...
}
```

This will make your styling changes on the tree nodes only affect that particular slideshow.

7.6 Slideshows

The text in slideshows should be complete sentences. Which means that nearly always, there should be a period at the end of the sentence. The only exception would be when a series of slides is building up a sentence, such as if one slide said "First we do this...", and then the following slide replaced it with "First we do this, then we do that."

7.7 Programming Exercises

To create a programming exercise, you will need to create/modify files on the front-end and others on the back-end:

- Front end:
  1. Go to OpenDSA/Exercises/ModuleName. ModuleName can be any of the modules in the Exercises directory (e.g. List, Binary, RecurTutor..etc)
  2. Create html file exercisename.html.
  3. Open the html file and modify the text of the following tag to have the problem statement:

     `<p class="problem" id = "test">`

     e.g. Complete the missing recursive call so that the following function computes something.
  4. Modify the text of the codeTextarea to have the code that required to be edited by the student:

     `<textarea  id="codeTextarea">`

     Example:

     ```
     int examplefunc(int i) {

      if (i > 0) {

       if (i % 2 == 1) {

        return i;

      }

      //<<Missing a Recursive call>>

     }

     }
     ```

5. Add a DOM variable to specify the programming exercise type (e.g. recursio, BinaryTree, List,..etc)

   Example:

   ```
   window.progexType= "recursion";
   ```

6. Open OpenDSA/config/ModuleName.json

7. Add the exercise in the exercises section as the following example:

   ```
   "recprogex1":{
   "long_name": "Recursion Programming Exercise Number or Description",
   "required": true,
   "points": 0.0,
   "threshold": 1.0}
   ```

8. Open OpenDSA/RST/en/ModuleName/ModuleName.rst

9. Add the following line so that the programming exercise appears in the lesson. As the following example:

   ```
   .. avembed:: Exercises/RecurTutor/recprogex1.html ka
   ```

10. Build the book on the front end:

1. Go to by the command CD OpenDSA/

2. Run the command: sudo make ModuleName

- Back end (Unit tests):

  1. Go to OpenDSA-server/ODSA-django/openpop/build/ModuleName
  2. Create a directory with the same name as the exercise name created on the front end (e.g. recprogex1)
  3. Create java file that will have the unit tests: exercisename.java (e.g. recprogex1.java)
  4. Open the exercisename.java.
  5. Name the class in the file as studentexercisename (e.g. studentrecprogex1). Note that the class should be missing its closing brace. The Python code on the back end will append that closing brace dynamically when the student submit his code. The Python code appends the function submitted by the student to the java code and add the closing brace dynamically.
  6. Create a function in the java file that returns the model answer.
  7. In the main function, create the code required for the unit tests and call the model answer function (e.g. int x= modelexercisefunction(i)).
  8. For each unit test, call both the model answer function and the function given to the student in the front end in:

     ```
     <textarea  id="codeTextarea">
     ```

     Example:

     ```
     examplefunc(int i)
     ```

  9. Compare both answers as follows:

     ```
     if (studentfunctionreturn(i) == modelexamplefunction(i)) SUCCESS = true;

     try{
     ```

```
PrintWriter output = new PrintWriter("output");

if (SUCCESS) {

output.println("Well Done!");
output.close();
}

else
{
output.println("Try Again! Incorrect Answer!");
output.close();
}

}
catch (IOException e) {
e.printStackTrace();
}
}
```

10. Note that: you should do the necessary logic to make sure that all the unit tests are correct. Also, you will not need to modify any of the Python files on the back end.

CHAPTER 8

Using OpenDSA to Create JSAV-based Proficiency Exercises

Place-holder for best practices and advice on writing JSAV-based proficiency exercises.

There is a tutorial for how to create a proficiency exercise at http://jsav.io/exercises/tutorial-exercise/.

8.1 Comparisons for gradeable steps

Often, correctness for a gradeable step is based on matching the current values within some data structure at each step. For example, you might want to compare the contents of an array at each step (in other words, what you check is that the user has put the right values into the array at each step).

Sometimes, what you want to compare is some other aspect of an element in the data structure. For example, perhaps what the student should be doing is clicking on certain array elements in order to highlight the correct ones. In this case, you are concerned with some property other than the value of the element. In such cases, the best approach is to add a class to the element through the click handler, as opposed to setting some physical property such as the background-color. Then, the gradeable step should be comparing that the classes on the elements in the user's copy of the data structure matches the classes on the elements in the model copy of the data structure.

CHAPTER 9

Using OpenDSA with the Khan Academy infrastructure

We use the Khan Academy (KA) infrastructure for writing questions. KA supplies some documentation at their wiki.

In the rest of this section, we will go through examples of the most common question types used in OpenDSA.

In general, each given question gets defined in its own HTML file (with an associated JavaScript file if necessary). This promotes maximum flexibility and reuse. New exercises are normally stored in `~OpenDSA/Exercises/Development` until they have been reviewed.

9.1 Summary Exercises

Individual questions, as appropriate, can be grouped together into "summary" exercise. These can in turn be grouped together into a broader summary, such as an end-of-chapter review. `~OpenDSA/Exercises/Background/IntroSumm.html` provides a typical example. If you are creating a new summary exercie, you can copy this file and start by changing the title on line 4. Note that the two library includes on lines 5 and 6 are needed for all Khan Academy exercises within OpenDSA. Replace lines 9-16 with your exercises, one per line, using the same format as these lines use. Note that `data-name` takes a file name using a path relative to the summary exercise.

9.2 Math and Variables

You can include math (in LaTeX notation) by bracketing it in `<code></code>`. See `~OpenDSA/Exercises/Background/MthBgMCQperm.html` for simple examples.

You can define variables to generate random values or compute results. `~OpenDSA/Exercises/List/ListOverhead.html` shows an example. It also shows simple use of JavaScript for computation, but anything more complicated than this should separate the JavaScript into a separate file (see examples below).

9.3 True/False Questions

`~OpenDSA/Exercises/Design/CompareTF2.html` is an example of a typical simple T/F question. Simply change the question in the "question" paragraph, set "solution" to be `True` or `False`, and set the hints.

ALL questions (aside from summary exercises) should have at least one hint!

`~OpenDSA/Exercises/Design/CompareTF1.html` shows use of simple alternate wording, called a "spin". Note the required change in line 2 to load "spin" support.

`~OpenDSA/Exercises/Background/SetTFequivrel.html` shows sophisticated use of variables to generate a variety of questions. It also shows use of `katex` functionality to support interpretation of math inside a string (not a typical problem that you will encounter).

9.4 Multiple Choice Questions

The simplest form of MCQ is shown in `~OpenDSA/Exercises/Background/IntroMCQeff.html`. Just change your title, your question, the text for the choices (add as many as you like), and the hints. This style of MCQ will show the answer string and the choices strings, all arranged in random order each time the question is presented. Note that the correct answer is **not** included as a choice. In contrast, T/F questions are a special type of MCQ where the question choices are given in the specified order (and the correct answer is therefore also one of the listed choices). `~OpenDSA/Exercises/Design/CompareMCQ4.html` is an MCQ with fixed choices.

`~OpenDSA/Exercises/Binary/HparentMCQ.html` shows a somewhat different approach. Line 18 has two key flags. `data-show` is set to 4, meaning that 4 choices will be shown. `data-none` is set to true, which means that the choice `None of the above` will always appear. Counting the correct answer, the four distractors, and `None of the above`, this means that there are six possible choices to display (`None of the above` will appear last, and three of the other five will appear at random). When the correct answer is not shown, `None of the above` becomes the correct answer. If a student chooses that, then the correct answer is shown after clicking the `Check answer` button.

9.5 Fill-in-the-Blank Questions

See `~OpenDSA/Exercises/Background/MthBgFIBlgbs1.html` for simple FIB formatting. It is important to understand that the KA framework provides support for processing certain types of student inputs. The simplest approach is to do an exact string match, as in this simple example. This is indicated by the fact that line 15 includes the tag `data-type="text"` (even though the student answer is actually a number, they have type it exactly this way). `~OpenDSA/Exercises/Background/AlgAnlsFIBsqrt.html` shows off two different features. It will allow the student to type either of two diffrent answers: the string “1/2” or anything that is the decimal equivalent to 0.5.

`~OpenDSA/Exercises/Background/MthBgFIBstckfn.html` uses custom JavaScript to interpret student answer. Note the specific syntax of lines 33-40 that you can use as a template. Again, anything more than a few lines of code should be separated into its own JavaScript file, as described next. For more details about what is going on here, see the KA wiki.

9.6 Using JavaScript and JSAV within the Khan Academy Framework

Using JavaScript in a KA exercise requires some non-standard processing (since these are not vanilla HTML files). We also include in our client-side framework special features to support including JSAV visualization elements.

`~OpenDSA/Exercises/AlgAnal/GrowthRatesPRO.html` and `~OpenDSA/Exercises/AlgAnal/GrowthRatesPRO` show a simple example. Note line 6 of the HTML file, where `CompareGrowth` is referenced. This is defining the name of the JavaScript file to include (normally the name has to match that of the HTML file, otherwise the search process will be looking in other places for the file). Note the syntax used on line 30 of `compareGrowth.checkAnswer`. This tells the exercise to look for the definition of function `checkAnswer` externally (in the JavaScript file). The first 4 and last 2 lines of this JavaScript file should be copied verbatim to your file, changing the name `compareGrowth` to the corresponding name that you use. It is relevant that the name matches the file name, with the first letter in lower case. Any functions or variables that are to be “exported” to the

HTML file need to be in the equivalent to the `compareGrowth` object. Other functions can be defined outside of that object in normal fashion.

`~OpenDSA/Exercises/AlgAnal/GrowthRatesPRO.html` and `~OpenDSA/Exercises/AlgAnal/GrowthRatesPRO` show a relatively simple example of using JSAV with the Khan Academy framework. Note that all of the following naming conventions should be observed unless you know what you are doing when you change it.

Line 7 of the HTML file requires that `jsav` be included, along with the name of the exercise file (which must be the same as the matching .js file to be included. Exercise file names (and their JavaScript file) must begin with a capital letter.

Any required CSS styling is included in the .html file for the exercise. Be aware that many elements have defined styles already that should not be overwritten without cause. `GrowthRatesPRO` is a rare exception in that it explicitly needs to define the JSAV array cells to be large. Most JSAV-based exercises have only a height value, and all CSS elements should include the file name to keep from colliding namespace with other parts of the page. Never adjust the width of the .jsavcanvas. You have no reason to make it narrower than the default, and you will probably break things if you make it wider.

The body tag (Line 12) must include the `data-height` and `data-width fields. Always set ``data-width="950"`. Make sure that the height can accommodate the hints when they are displayed.

The `div` tag on line 44 has an ID field whose name is the exercise name with "p" added at the end. The `div` tag on line 52 has an ID field whose name is the exercise name. Any variables or functions imported from the JavaScript is preceeded with the exercise name, but with the first letter in lower case. Lines 54-63 should use this syntax, with the exercise name changed as appropriate.

Moving to the JavaScript file, we first see that `window` is always tagged as a global. Any other global packages (in this case, `katex` would appear there as well. We always use the functional version for the `"use strict"` directive, so as not to break other JavaScript loaded on the page (in particular, the KA framework code won't work under strict rules).

Any functions or variables to be exported to the HTML file **must** be declared inside an object that uses the exercise name, with the initial letter changed to lower case. We must use the declaration syntax shown for the declarations in the object. Any other functions and variables that we want to create, outside the scope of the object, can use your preferred JavaScript declaration syntax. Note the last two lines of the file, where the object name is exported.

The JSAV object **must** be declared with the exercise name for its parameter. It is petty much universal that the JSAV objects appear as a static display (that is, not a JSAV slideshow). Lines 86-90 show the standard format for this section, beginning with creating the JSAV object, followed by setting out any JSAV objects (this one is pretty simple in that there is only a single JSAV array), followed by the `displayInit` and `recorded` to get the elements displayed on the HTML page.

9.7 Naming Conventions

Summary exercises should **always** be given names of the form `<topicname>Summ.html`.

T/F questions should have file names as `<topicname>TF<question>.html`. Multiple choice questions should be named as `<topicname>MCQ<question>.html`. Fill-in-the-blank questions should be named as `<topicname>FIB<question>.html`. It is bad style to use counts (like 1, 2, 3) for the `<question>` part in the name for a series of questions.

Proficiency exercises should be named as `<topicname>PRO.html`. A proficiency exercise is something where the student must manipulate a data structure (including any exercise where the user interface for answering the question is to click or drag JSAV objects), or sometimes work a problem to reach an answer. In the case where students do a computation to complete a fill-in-the-blank problem, this might be a matter of judgement on how to name it. Generally, a fill-in-the-blanks question that generates random problem instances and is presented as a stand-alone exercise to students should probably be named as a proficiency exercise.

9.8 Common Errors

Your exercise might generate a console error that looks like:

```
Error while evaluating var#JSAV
khan-exercise.js:359 TypeError: Cannot read property 'id' of undefined(...)
```

This is a common problem. Nearly always, it means that either you forgot to include `jsav` in the `data-require` field of the `<html>` tag, or else you got the names wrong somewhere for the divs.

CHAPTER 10

Client-side Development

10.1 AV Developer Responsibilities

The OpenDSA client-side framework automatically handles as much logging as possible. However, developers must write their AVs to be compliant with the framework to ensure proper logging (**Note**: Non-compliant AVs will fail silently, so follow the instructions closely and test your AV to ensure the data you expect to be logged is logged).

- Follow standard HTML layout as close as possible (use templates)
  - Ensures proper CSS and scripts are loaded including: `JSAV.css`, `odsaAV-min.css`, `jquery.min.js`, `jquery-ui.min.js`, `jsav-min.js`, `odsaUtils-min.js`, `odsaAV-min.js`
  - Provides a standard appearance for all AVs
  - Makes maintenance and systematic changes easier
- Use descriptive IDs to reference page elements
  - All IDs on a single page must be unique, allowing reliable access to a specific element
  - The framework will automatically attach loggers to buttons and links (unless they appear inside a contain with a class that matches '.*jsavw*control.*') and the data these loggers collect identifies interactive elements based on their ID. Using descriptive IDs will make data analysis much easier.
  - Do not place elements inside a contain with a class that matches `'.*jsav\w*control.*'`. JSAV controls appear within these containers but are logged by a different mechanism so the automatic button and link loggers ignore the contents of these containers.
- Make JSAV exercise options configurable (if necessary and desired)
  - If you do not change the default values of `JSAV_EXERCISE_OPTIONS`, you do not have to do anything.
  - However, if you do change the default values of `JSAV_EXERCISE_OPTIONS`, you must call `ODSA.UTILS.parseURLParams()` in order to make the exercise options configurable from the config file. Conversely, if you want to prevent the configuration process from overriding the grading options for your exercise, you can change or reset the defaults and not call this function.
  - Developers should initialize configurable variables like this: `var variable = (params.<variable>) ? params.<variable> : default_value;`, so that if the appropriate parameter is passed to the exercise it will be used, otherwise the exercise will default to a reasonable value
- Attach JSAV array click handlers through JSAV rather than jQuery

  - Example: `theArray.click(function(index){...});` rather than `$('#arrayId').on('click', ".jsavindex", function(){...});`

- Log initial exercise state
  - If the only input to your AV is an array of input values (such as most of the Sorting AVs), you can use `ODSA.AV.processArrayValues()` which will automatically log the array values used to initialize your AV
  - If `ODSA.AV.processArrayValues()` does not fit your needs, you can all `ODSA.AV.logExerciseInit()` explicitly. Refer to the function documentation in `odsaAV.js` for more information on logging conventions.
- Take advantage of macro-logging
  - As a developer, if you feel something is important to log (whether its the state of the exercise, some sort of interaction or simply a comment) you can use either `ODSA.UTILS.logUserAction()` or `ODSA.UTILS.logEvent()`. Refer to the function documentation in `odsaUtils.js` for more information on how to use these functions.
  - Allows developers to describe what is happening at a given step to make it easier when analyzing problem steps to identify what step the students are missing

10.2 Internationalization Support

OpenDSA supports a sophisticated internationalization framework that attempts to make it as easy as possible to support compiling textbook instances in various (natural) languages. The configuration system allows a book compiler to specify the language of choice, and the system will take module versions in the target language whenever available (the fallback language is English).

Like every other aspect of internationalization, we define the language using the two-letter ISO 639-1 language codes.

Someone creating a new book instance would use the 'lang' variable in the configuration file to define their book language. But if you want to add support to the OpenDSA system to support a new language, then you will need to provide the necessary strings in the target language. Here is a guide to how you would provide that information.

It helps to understand that the Sphinx compiler itself has its own translation support, which affects some of the strings that appear on an OpenDSA page. Just telling Sphinx what language that you want to use will cause those strings that Sphinx controls to be translated. This is done by the configuration system when the configuration file tells it to use a particular language. A list of languages supported by sphinx can be found at http://sphinx-doc.org/config.html#confval-language.

Translation is controlled by the file `tools/language_msg.json`. Each language is represented by its code in language_msg.json. Make sure that a translation is available in language_msg.json file before asking the configuration system to create a book in that language.

The terms for each language are grouped in two categories within `language_msg.json`:

- **`jinja` for the terms that will be added inside the configuration** file. They will be passed by Sphinx to the templating system (jinja + haiku).
- **`js` for the terms processed by the `odsaMOD.js` library, and** injected while the page is loading.

Here is the structure for language_msg.json:

```
{
  "en"{
    "jinja": {
      "term1": "en_term1",
      ...
    },
    "js": {
      "term2": "en_term2",
      ...
    }
  },
  "fi"{
    "jinja": {
      "term1": "fi_term1",
      ...
    },
    "js": {
      "term2": "fi_term2",
      ...
    }
  }
}
```

The gradebook text strings are taken from `RST/<lang>/Gradebook.rst`.

The book configuration program will read the language variable. If a translation for the entered language is not available, the default language English is used.

Individual AVs and exercises support internationalization through the use of an associated `.json` file that provides the various translation text for all strings that appear in the AV. JSAV provides translations to many languages for its infrastructure strings.

10.3 Glossary Concept Map Definition

OpenDSA supports displaying glossary terms as a **concept map**. The relationship between terms are specified in the `Glossary.rst` file, and consist of the following elements added below the term we are defining:

- **`:to-term:` followed by the related term. Ideally, the related term should be also** defined in the glossary file, but is is not mandatory.
- `:label:` followed by the linking phrase decribing the relationship between the two terms.

Here is an example of a relationship definition between the terms `graph` and `vertices`:

```
graph
    :to-term: vertices :label: contains

    A :term:`graph` :math:`\mathbf{G} = (\mathbf{V}, \mathbf{E})` consists
    of a set of :term:`vertices` :math:`\mathbf{V}` and a set of
    :term:`edges` :math:`\mathbf{E}`,
    such that each edge in :math:`\mathbf{E}` is a connection between a
    pair of vertices in :math:`\mathbf{V}`.
```

10.4 Client-side API

10.4.1 ODSA.AV

- **logExerciseInit** - generates an event which is used to log the initial state of an AV or exercise

  - Captures the state of the exercise at the beginning (such as the numbers in an array) which allows us to put the later operations we log in context. For example, its all well and great to know the user clicked on index 4 of an array but if we don't know the randomly generated numbers in the array the operations we log won't make much sense.
  - The generated event uses the same channel as JSAV events and is therefore received by the existing listener. This function is NOT dependent on the JSAV framework.
- **awardCompletionCredit** - generates an event which triggers the framework to give a user credit for an exercise
  - This function is designed to be used when an exercise doesn't really have a score but must be completed (like the calculator or performance exercises). Developers should call this function in their code when a student has reached a state where the developer believes they should receive credit.
  - The generated event uses the same channel as JSAV events and is therefore received by the existing listener. This function is NOT dependent on the JSAV framework.
- **initArraySize(min, max, selected)** - initializes the arraysize drop down list with the range of numbers from `min` to `max` with `selected` selected
- **reset(flag)** - resets the AV to its original state
  - The `reset()` function works by saving the HTML from the `avcontainer` element on page load and using it to replace the HTML in the `avcontainer` when reset it called. When JSAV is initialized it alters the contents of the container, after the HTML has been saved. When JSAV is initialized on page load but never reinitialized, the first reset clears the elements JSAV generated, breaking the AV. Using this `reset()` method, JSAV must be reinitialized after each reset in order for the AV to function properly. We recommend reinitializing JSAV after calling `ODSA.AV.reset(true)` in the `runit()` method.
  - The `runit()` method should call `ODSA.AV.reset(true)` to ensure the avcontainer is cleared and ready for the next instance.
- **processArrayValues(upperLimit)** - validates the array values a user enters or generates an array of random numbers if none are provided
- **sendResizeMsg()** - forces the AV to send a message to the parent page containing the height and width of the rendered AV. This function is called automatically when the AV is loaded or reset, but can be explicitly called by developers if their AV changes size during its operation.

10.4.2 ODSA.MOD

- **serverEnabled()** - returns whether or not the backend server is enabled
- **inDebugMode()** - returns true if localStorage.DEBUG_MODE is set to true
- **getBookID()** - returns a SHA1 hash of the book URL as a unique identifier
- **getUsername()** - returns the username stored in local storage
- **getSessionKey()** - returns the session key stored in local storage
- **userLoggedIn()** - returns whether or not a user is logged in
- **getJSON(data)** - converts the input string to a JSON object, if given a JSON object, returns it
- **logUserAction(type, desc, exerName, eventUiid)** - logging function that takes the event type, a description of the event and the name and uiid of the exercise with which the event is associated
- **logEvent(data)** - flexible logging function that appends whatever data is specified to the event log, provided it meets the criteria for a valid event
- **sendEventData()** - flushes the buffered event data to the server

- **getType()** - returns correct object type information (replaces broken functionality of 'typeof')
- **roundPercent(number)** - rounds the given number to a max of 2 decimal places

10.4.3 ODSA.UTILS

- **STATUS** - pseudo-enumerated variable used to define the different states of proficiency
- **getProficiencyStatus(name, username, book)** - returns whether or not local storage has a record of the given user being proficient with the given exercise or module in the given book
- **syncProficiency()** - queries `getgrade` endpoint to obtain proficiency status for all exercises and modules
- **parseURLParams()** - parses parameters from the URL, sets `JSAV_OPTIONS` and `JSAV_EXERCISE_OPTIONS` if applicable and stores the remaining options in a global `PARAMS` object for use by the module or exercise

10.5 Tips and Tricks

10.5.1 Truthy and Falsy

Be aware that values in JavaScript will not always evaluate the way you expect when used in conditionals. When comparing objects use **strict equal** (`===`) and **strict not equal** (`!==`) to ensure values are compared by type and value. When testing whether a variable contains useful information you can generally use the value inself in the conditional, i.e. `if (testCondition) {...}`. While this is 'sloppy', it works unless you expect a `0`, `false` of `""` to be valid. If you want a more formal test, you can use `typeof testCondition === "undefined"`. This expression will be true only if `testCondition` has never been assigned a value.

For more information see Truthy and Falsy: When All is Not Equal in JavaScript.

10.5.2 HTML5 postMessage

We have no guarantee that content embedded in iFrames (such as AVs and Exercises) will be hosted on the same domain as the modules. In order to create a robust application communication between the parent and child pages should take place using `postMessage` rather than referencing elements or functions through the `contentDocument` or `contentWindow.document` of the iFrame element or `window.parent` or `window.top`.

10.5.3 Encapsulation

You should always wrap your JavaScript code in an anonymous function to prevent the DOM from getting cluttered and to prevent outside access to specific data or functions. All functions and global variables defined within an anonymous function are visible to each other and can be used normally. However, sometimes you will need to define a publically accessible function that interacts with functions you wish to keep private. The simplest way to do this is to write your JavaScript as normal within an anonymous function and then assign specific "public" functions to be properties of the `window` object. For example:

```
$(document).ready(function () {
  var privateData = 0;

  function privFunct() {
    alert('ODSA private function');
  }
  function publicFunct() {
    privFunct();
  }

  var AV = {};
  AV.publicFunct = publicFunct;
  window.AV = AV;
});
```

Another alternative is:

```
$(document).ready(function () {
  var AV = {};

  function privFunct() {
    alert('ODSA private function');
    AV.publicFunct();
  }

  AV.publicFunct = function() {
    alert('ODSA publicFunct');
  }

  AV.callPrivFunct = function() {
    privFunct();
  }

  window.AV = AV;
});
```

In both of these examples, `publicFunct()` can be referenced outside the anonymous function using `AV.publicFunct()` (or `window.AV.publicFunct()`). We prefer the first method because it looks more like a standard JavaScript file, internal function references are simpler, and its easy to add all the public functions in one place, giving the developer greater control over what they make public.

Be sure not to overwrite any existing namespaces (such as window.ODSA which is used by the OpenDSA framework)!

10.6 Troubleshooting

10.6.1 jQuery Selectors

jQuery selectors can be useful, but do have some limitations. For instance, when using jQuery to reference an element by ID, the ID cannot contain specific characters such as a period, a plus sign or spaces. While its better to avoid them if possible, if you find that you must use these or other invalid characters, use `$('[id="' + objID + '"]')`.

10.6.2 Proficiency Exercises

- If your AV doesn't show up immediately but shows up as soon as you advance the slideshow, make sure you ran: `jsav.displayInit();`
- If you are having difficulties with variables managed by JSAV
  - Make sure you use `.value()` to access the variables value, otherwise you get an object rather than the string or number you most likely want

  - Make sure you use `.value(newValue)` to change the value of the variable, assignment using '=' doesn't work

- If your fixState function successfully changes the state of everything, but says you are getting all subsequent correct answers wrong and undoing everything to the state where you first made a mistake, make sure you are calling `exercise.gradeableStep();`

ReST Extensions

The following custom ReST extensions have been created for the OpenDSA project.

The documentation here presents all of the options associated with each directive, as if the directive were to appear for direct processing by Sphinx. However, OpenDSA modules are intended to be pre-processed by a configuration script that dynamically loads in additional information to tailor modules for specific eTextbook instances. In particular, information related to grading of embedded exercises should be controlled by the configuration files. See *Compiling Book Instances: OpenDSA Configuration* for details. Thus, a number of the directive options, while documented, are labeled as not being set manually (i.e., by the module author) within the ReST file. You just leave those options out when you create your module, and specify them instead in your configuration file.

11.1 avembed

NAME avembed - embed an AV or other HTML page inside a ReST document.

SYNOPSIS:

```
.. avembed:: {relative_path} {type}
   [:exer_opts: {string}]
   [:long_name: {string}]
   [:module: {string}]
   [:points: {number}]
   [:required: true|false]
   [:showhide: show|hide|none]
   [:threshold: {number}]
```

DESCRIPTION

`.. avembed:: {relative_path} {type}` `{relative_path}` is the relative path (from the OpenDSA home directory) to the embedded page. `{type}` is the type of embedded exercise, one of the following:

- **ka** - Khan Academy style exercises
- **pe** - OpenDSA proficiency exercises
- **ss** - slideshows
- **dgm** - JSAV-based diagram

`:exer_opts: {string}`

A URL-encoded string of configuration options to pass to exercises. **Added automatically by the configuration process, do NOT add manually.**

`:long_name: {string}`

Long name for the embedded object. The "short" name is the file name. **Added automatically by the configuration process, do NOT add manually.**

`:module: {string}`

The name of the module on which the AV is embedded. **Added automatically by the configuration process, do NOT add manually.**

`:points: {number}`

Number of points this activity is worth. **Added automatically by the configuration process, do NOT add manually.**

`[:required: true|false]`

Whether this activity is required for module credit. **Added automatically by the configuration process, do NOT add manually.**

`:showhide: show|hide|none` Include a button to show or hide the embedded content. The options are `show` to have the content visible when the page is first loaded or `hide` to have it hidden on page load. **Added automatically by the configuration process, do NOT add manually.**

`[:threshold: {number}]`

Threshold number of points required for credit. **Added automatically by the configuration process, do NOT add manually.**

NOTES The `.. avembed::` directive fetches the AV's information (width and height, etc.) from its XML description file. This XML file is located in the directory `xml` contained within the same directory as the AV. If the AV is named `fooAV.html` then the XML file must be `xml/fooAV.xml`. The path to the OpenDSA top-level directory should be set within `conf.py` in the source directory.

11.2 avmetadata

NAME avmetadata - metadata information associated with this module.

SYNOPSIS:

```
.. avmetadata::
   :author: {string}
   :prerequisites: {list of module_name}
   :topic: {string}
   :requires: {string}
   :satisfies: {string}
```

DESCRIPTION

`:author: {string}` Module author's name.

`:prerequisites: {list of module_name}` A comma-separated list of zero or more `module_name`. These represent the prerequisites for this module.

`:topic: {string}` The topic covered by this module.

`:requires: {string}` A semi-colon delimited list of topics students are expected to know prior to completing the module

`:satisfies: {string}` A semi-colon delimited list of topics covered in this module that satisfy prerequisite knowledge requirements of other modules

11.3 codeinclude

NAME codeinclude - displays code snippets within the eTextbook.

SYNOPSIS:

```
.. codeinclude:: {relative_path}
   [:tag: {mytag1} [, {mytag2}, ...]]
```

DESCRIPTION `.. codeinclude:: {relative_path}`

Include the contents of the file located at `{relative_path}`. If the path is relative to `code_dir`, that specific file will be loaded. However, if the path is relative to a code language directory in `code_dir`, the directive will attempt to load the file in all the languages (specified in `code_lang`) in a tabbed display if `tabbed_codeinc` is `True` and only the language with highest precedence if `tabbed_codeinc` is `False`. Convention dictates that the file extension be omitted when using the second option, however, the directive will automatically strip the file extension if one is provided.

`:tag: {my_tag}`

Optionally, a tag or a comma separated list of tags can be specified. These tags must appear inside the source code file as specially formatted comments that delimit the block(s) of code that will be included. If tags are used, then only the code block(s) with the tags will appear. If multiple tags are used, then the multiple blocks will appear as though they were one continuous block of code without the intervening code that does not appear with the tags. If additional tags are hierarchically embedded within a tag block, then those tags will not appear (but the code will). Note that the source code must format the tags correctly, as:

```
/* *** ODSATag: my_tag *** */
/* *** ODSAendTag: my_tag *** */
```

`:lang: {code_language}`

Specifies the language of the code to be loaded (overrides multiple language loading).

NOTES The `codeinclude` directive closely matches the standard ReST directive `literalinclude`.:

```
.. codeinclude:: {relative_path}
   [:tag: my_tag]
```

will (logically) map to::

```
.. literalinclude:: <relative_path>
   :start-after: /* *** ODSATag: my_tag *** */
   :end-before: /* *** ODSAendTag: my_tag *** */
```

11.4 inlineav

NAME inlineav - used to embed an AV (in particular “slideshows”) into the document

SYNOPSIS:

```
.. inlineav:: {avId} {type}
   [:output: show|hide]
   :points: {number}
   :required: true|false
   :threshold: {number}
   :align: left|right|center|justify|inherit
```

DESCRIPTION `.. inlineav:: avId type`

Create a container for an inline AV with the given ID and type. If the type is `ss` a slideshow will be created and if it is `dgm` a diagram will be created. inlineav diagrams behave like typicall reStructuredText figures. Cross reference target and caption are declared using the standard syntax.

`<type>` **is automatically appended to the inlineav directive by the configuration process and should not be added manually.**

`[:output: show|hide]`

If the AV is a slideshow, controls whether or not the message box is displayed Note the 'output' argument is only valid for slideshows.

`:long_name:`

Long-form name for a slideshow object. **Added automatically by the configuration process, do NOT add manually.**

`:points: {number}`

Number of points this activity is worth. **Added automatically by the configuration process, do NOT add manually.**

`:required: true|false`

Whether this activity is required for module credit. **Added automatically by the configuration process, do NOT add manually.**

`:threshold: {number}`

Threshold number of points required for credit. **Added automatically by the configuration process, do NOT add manually.**

`:align: left|right|center|justify|inherit`

The alignment of the caption within the page.

11.5 numref

NAME numref - adds numbered cross references to modules.

SYNOPSIS:

```
:numref: {caption} <{reference_label}>
:numref: {reference_label}
```

DESCRIPTION `:numref: {caption} <{reference_label}>`

A custom interpreted text role. `numref` adds numbered cross references within ODSA documents.

`{caption}`

Text to be displayed next to the numbered reference.

`{reference_label}`

Reference name (unique) of the referenced object. Should be enclose in brackets (<>) when a caption is provided. It is specified via the standard ReST referencing mechanisms.

NOTES The ODSA preprocessor creates a table of all referenced objects with numbers and writes it into a file that is read by the `numref` role. When referencing equation (declared with `math` directive), 'equation-' need to be added in front of the label to work, eg to reference the equation with label 'sum2' you write `:numref:`<equation-sum2>``

WARNING: We now consider it a violation of best practice to reference a module from another module. This is because OpenDSA is a collection of materials that can be combined in various ways. For this reason, use of `numref` has been phased out.

11.6 chap/numchap

NAME chap/numchap - adds a reference to the first (introduction) module of a chapter.

SYNOPSIS:

```
:chap: {chapter_name}
:numchap: {chapter_name}
```

DESCRIPTION `:chap: {chapter_name}`

A custom interpreted role that adds the chapter name as the label for a link to the first module of the indicated chapter.

`:numchap: {chapter_name}`

A custom interpreted role that adds a chapter number as the label for a link to the first module of the indicated chapter.

`{chapter_name}`

The name of the chapter. It should be identical (case sensitive) to the one specified in the json configuration file.

11.7 showhidecontent

NAME showhidecontent - creates a section of text that can be hidden or displayed.

SYNOPSIS:

```
.. showhidecontent:: {section_id}
   [:long_name: {string}]
   [:showhide: show|hide|none]
```

DESCRIPTION

`.. showhidecontent:: {section_id}` `{section_id}` is a string used to identify the section in the configuration file. Ideally, it should be descriptive and in camel-case, because if `long_name` is omitted, `section_id` will be converted to a space-delimited string and used in its place

`:long_name: {string}`

The display name for the section that will appear on the showhide button (if applicable). If omitted, the `section_id` will be converted from camel-case to a space-delimited string and used in its place **Added automatically by the configuration process, do NOT add manually.**

:showhide: show|hide|none If `show` then display a button to show or hide the section and make the section visible on page load. If `hide` then display the button, but hide the section on page load. If `none` or if the option is omitted then the section will be displayed with no button **Added automatically by the configuration process, do NOT add manually.**

11.8 TODO

NAME TODO - adds a todo box in the output HTML file, and is also used by the ODSA preprocessor script to create a separate HTML page containing the collated list of desired AVs and Exercises. (NOTE: Can also be called as `todo`.)

SYNOPSIS:

```
.. TODO::
   [:type: {type label of the desired artifact}]
```

DESCRIPTION

`.. TODO::`

Within the module, this behaves like the standard Sphinx TODO (or todo) directive. As with the standard TODO directive, the author should then include (indented) text that describes the task to be done. The ODSA version will in addition create a separate page TODO.html that includes a listing of all TODO blocks from all of the modules.

`:type: {type label of the desired artifact}`

The type of the desired artifact (AV, Proficiency Exercise, etc). This is just a label, so it can be anything. Each separate label will collate together all TODO entries with that label on the TODO.html page.

NOTES The ODSA preprocessor collects the descriptions (i.e., the text that follows the TODO directive) from the complete collection of RST files to create a separate TODO.rst file that lists all the desired AVs and Exercises grouped by type. The TODO.rst file should be included in the index.rst file to be part of the table of contents for the eBook.

11.9 odsalink

NAME odsalink - adds the code to include a CSS file in the HTML output file.

SYNOPSIS:

```
.. odsalink:: {path to file}
```

DESCRIPTION `.. odsalink::` The directive injects the code to include a linked file in the outputted HTML files. It gets the path to ODSA directory from the `odsa_path` variable in the `conf.py` file.

`{path to file}` The path (relative to ODSA directory root as defined by the `odsa_path` variable in the `conf.py` file) to the linked file to be include.

NOTES The directory containing the file to be included should be hosted within the ODSA directory. Example, if `odsa_path` is defined to be `..\..\..`, then

`.. odsalink:: JSAV/css/JSAV.css`

will produce

```
<link href="../../../JSAV/css/JSAV.css" rel="stylesheet" type="text/css"
/>
```

in the HTML output file.

11.10 odsascript

NAME odsascript - adds the code to include a script file in the HTML output file.

SYNOPSIS:

```
.. odsascript:: {path to file}
```

DESCRIPTION `.. odsascript::` The directive injects the code to include a script file in the outputted HTML files. It gets the path to ODSA directory from the `odsa_path` variable in the `conf.py` file.

`{path to file}` The path (relative to ODSA directory root as defined by the `odsa_path` variable in the `conf.py` file) to the script file to be include.

NOTES The directory containing the file to be included should be hosted within the ODSA directory. Example, if `odsa_path` is defined to be `..\..\..`, then

`.. odsascript:: JSAV/build/JSAV-min.js`

will produce

`<script type="text/javascript" src="../../../JSAV/build/JSAV-min.js"></script>`

in the HTML output file.

11.11 odsafig

NAME odsafig - provides the ability to specify caption alignment to figures.

SYNOPSIS:

```
.. odsafig:: {path to image}
   :capalign: left|right|center|justify|inherit
```

DESCRIPTION `.. odsafig::` The directive behaves exactly as the standard `.. figure::` directive. It allows you to specify the positioning of figure caption on the page

`:capalign: left|right|center|justify|inherit` The alignment of the caption on the page.

NOTES The directive closely matches the standard ReST `figure` directive. The only addition is the `:capalign:` argument.

11.12 odsatab

NAME odsatab - provides the ability to create tables (with `math` directive) that behave like figures. Caption is display above the table, and the position of the caption can be specified by the user.

SYNOPSIS:

```
.. odsatab::
   :capalign: left|right|center|justify|inherit
   :align: left|right|center
```

DESCRIPTION `.. odsatab::` The directive allows the user to create tables using `math` directives. The directive numbers tables and allows numbered cross refences. It allows users to specify the positioning of the table and the table caption on the page

`:capalign: left|right|center|justify|inherit` The alignment of the caption on the page.

`:align: left|right|cente` The alignment of the table on the page.

NOTES The first paragraph of the directive content is used as table caption.

11.13 odsatoctree

Specialized version Sphinx `toctree` directive. It is used when a chapter has the optional `hidden` field to `true`. The Modules in the chapter will not be visible in the table of content. **It is added automatically by the configuration process, do NOT add manually.**

11.14 ref

We have modified the Sphinx `ref` directive to better support the fact that eBook instances can vary with respect to whether given modules are included or not.

NAME ref - Creates a hyperlink to a module, label, or a glossary term.

SYNOPSIS:

```
:ref:`my anchor text <label>`
:ref:`my anchor text <glossary term> <label>`
```

DESCRIPTION `my anchor text`

The anchor text for the hyperlink.

`<label>`

Module name or some label within a module. If it is a module name, then `ref` links to the module. If it is a label (such as for an Example), then the directive links to that point in the module. If `<label>` does not exist, then the directive shows only the anchor text (in normal font, as though no reference were being made).

`<glossary term>`

If `<label>` does not exist and the `<glossary term>` is given, then the hyperlink directs to the `<glossary term>` entry in the glossary.

11.15 topic (special case)

The syntax of the `topic` directive is not changed in OpenDSA. We use this directive to display `examples, tables, and theorems`. To insert an example in your module, use the keyword `Example` as the topic title.

To insert a theorem in your module, use the keyword `Theorem` as the topic title. The example/table/theorem can be referenced using the standard Sphinx mechanism. For a numbered reference, use the `:num:` directive.

EXAMPLE:

```
(1) to add  an example with an anchor
.. _example1:

.. topic:: Example

This is our first example


(2) to reference the example
See Example :num: `Example #example1`.
```

CHAPTER 12

The Document Processor

The OpenDSA textbook compilation pipeline includes custom preprocessing of module files into compileable ReStructuredText source. The main motivation for using our own document pre-processor was to support integration beyond the file level in ways that Sphinx does (or at least, did) not do. This includes the ability to number document objects (figures, and tables, and equations), and display numbered references. When we started the OpenDSA project, DocUtils did not providie such features. Some of the pre-processor features might be added over time to Sphinx, in which case we might eventually remove them from the pre-processor. You can view the DocUtils To Do list at http://docutils.sourceforge.net/docs/dev/todo.html.

12.1 Overview

The document processor works as a three-pass compiler. The first two passes are executed on `rst` files before running Sphinx, and the last pass is run against `html` files produced by Sphinx. The process results in three files, two containing ducuments and objects numbers and one to check if the document has been modified. All global variables are declared in a separate file (config.py).

12.2 First Pass

INPUT Modules as `rst` source files.

OUTPUT A file JSON (page_chapter.json) containing a dictionary of modules and their associate chapter.

DESCRIPTION During the first pass, the document processor creates a dictionary of the highlest level elements in the document (modules). The dictionary contains tuples defined as `(module_name, [chapter_name, chapter_number])`.

12.3 Second Pass

INPUT Modules as `rst` source files.

OUTPUT A JSON file (table.json) containing a dictionary of all documents objects and their appearance number.

DESCRIPTION During the second pass, the document processor creates a dictionary of all the objects inside modules. The appearance number is the concatenation of `chapter_number`, `module_number`, and `object_number`. The dictionary contains tuples defined as `(object_name, appearance_number.)`.

12.4 Integration with Sphinx

The `numref` (*numref*) directive adds numbers to document objects (figures, tables, and equations) to the output of the document preprocessor and uses it as hyperlink text for cross referencing.

12.5 Third Pass

INPUT Modules as `html` files generated by Sphinx.

OUTPUT Modified `html` files with an updated table of contents and navigation bar, and section numbers augmented with a chapter number prefix.

DESCRIPTION During the third pass, the document processor parses the html files and replaces headers and section numbers as appropriate from the dictionaries created during the first two passes. Since our processor does not modify the Sphinx document tree, we have to modify `html` files to replace the raw Sphinx section number with our own numbering scheme. This phase applies only to the Table Of Content, the navigation bar, page headers, and sections. The document processor perform a third pass only if the html file has been modified by Sphinx. The file `count.txt` stores the latest modification times for the html files.

12.6 Where things are

There are many files that affect the eventual HTML output. Here is a list of places to look if you are trying to make changes.

OpenDSA/RST/source/_themes/haiku/basic/layout.html

OpenDSA/RST/source/_themes/haiku/static/haiku.css_t

OpenDSA/RST/preprocessor.py

OpenDSA/RST/ODSAextensions

OpenDSA/tools/configure.py

CHAPTER 13

OpenDSA Backend Installation and Setup

The OpenDSA "backend" provides support for collecting student scoring data (in a MySQL database), logs user interaction details with OpenDSA content for analysis purposes, and provides tools for instructors to manage the student score data. Most OpenDSA developers do **not** need to worry about intalling a copy of the OpenDSA back end. Instructors who would like to use OpenDSA will probably want to contact the OpenDSA team about getting hosting support, rather than set up their own server.

13.1 Windows

1. Install Python
2. Install python setuptools
    - Download the executeable from http://pypi.python.org/pypi/setuptools
3. Install Django:

```
tar xzvf Django-1.4.1.tar.gz
cd Django-1.4.1
python setup.py install
```

4. Install MySQL Server

- Start a MySQL command prompt and then do:

```
CREATE DATABASE <database_name>;
GRANT ALL ON <database_name>.* TO '<database_user>'@'localhost' IDENTIFIED BY '<database_user_pa
exit
```

See "Both" section for remaining instructions

13.2 Linux

1. Install Python:

```
sudo apt-get install python
```

2. Download and install Django:

```
tar xzvf Django-1.4.1.tar.gz
cd Django-1.4.1
python setup.py install
```

3. Install python setuptools:

   ```
   sudo apt-get install python-setuptools (easy_install)
   ```

4. Install MySQL Server:

   ```
   sudo apt-get install mysql-server
   ```

   - Start mysql prompt and do:

     ```
     mysql -u root -p
     < Enter your password when prompted >
     CREATE DATABASE <database_name>;
     GRANT ALL ON <database_name>.* TO '<database_user>'@'localhost' IDENTIFIED BY '<database_us
     exit
     ```

See "Both" section for remaining instructions

13.3 Both

1. easy_install MySQL-python

- If this doesn't work in Windows, you can install from an EXE - https://code.google.com/p/soemin/downloads/detail?name=MySQL-python-1.2.3.win32-py2.7.exe

2. Install required modules, (remove `sudo` for Windows):

   ```
   sudo easy_install oauth2
   sudo easy_install simplejson
   sudo easy_install feedparser
   sudo easy_install icalendar
   sudo easy_install mimeparse
   sudo easy_install python-dateutil
   sudo easy_install django-tastypie
   sudo easy_install html5lib
   ```

3. Install memcache:

   ```
   pip install python-memcached
   ```

4. Install user agent (http://pypi.python.org/pypi/django-user_agents):

   ```
   pip install pyyaml ua-parser user-agents
   pip install django-user-agents
   ```

5. Install OpenDSA-server:

   ```
   git clone https://YOURGITHUBID@github.com/OpenDSA/OpenDSA-server.git
   cd OpenDSA-server/ODSA-django
   ```

   - Change values in settings.py file
     - 'ENGINE': 'django.db.backends.mysql', # Add 'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or 'oracle'.
     - 'NAME': '<database_name>', #g3et_path('test.db'), # Or path to database file if using sqlite3.
     - 'USER': '<database_user>', # Not used with sqlite3.
     - 'PASSWORD': '<database_user_password>', # Not used with sqlite3.

   - 'HOST': '', # Set to empty string for localhost. Not used with sqlite3.
   - 'PORT': '', # Set to empty string for default. Not used with sqlite3.
   - For both values under 'TEMPLATE_LOADERS = (' change 'load_template_source' to 'Loader'
   - Update BASE_URL to have IP and port of Django server (optional?)
     * BASE_URL = "<IP>:<PORT>"
     * Ex: BASE_URL = "127.0.0.1:8000"
6. Create an empty file named daily_stats.json, inside the "media root" directory specified in settings.py file
7. `python manage.py syncdb`
8. Create an administrator (superuser) account when prompted
9. `python manage.py runserver 0.0.0.0:8000`
10. In your web browser, go to: http://127.0.0.1:8000/admin/

13.4 SSL Certificates

An OpenDSA installation might encounter the symptom that some students cannot log on. If this is happening, it might be caused by a problem with your SSL certificate chain. If this is the cause, then hopefully you will be able to tell because either the browser console window or the Network tab under Firebug might indicate a message such as "Failed to load response data network error, ERR_INSECURE_RESPONSE".

The cause for this condition is that either the root certificate is missing in your Apache configuration, or else the certifciate signature chain is broken. In our experience, this can easily happen in a University setting.

You can use online tools to help diagnose SSL installation issues. See https://www.digicert.com/help/ or https://www.sslshopper.com/ssl-checker.html. Then, if necessary you will have to update your Apache configuration file (such as /etc/apache2/sites-enabled/default-ssl). Depending on your problem, you might need to:

- Add the root certificate by setting the `SSLCACertificateFile` variable (see http://httpd.apache.org/docs/2.2/mod/mod_ssl.html#sslcacertificatefile).
- Specify the certificate signature chain file with `SSLCertificateChainFile` (see http://httpd.apache.org/docs/2.2/mod/mod_ssl.html#sslcertificatechainfile). If your certificate vendor did not provide you with a single chain file, you might have to concatenate all the intermediate certificates into one file.

13.5 Caching

If you frequently update your OpenDSA's files, you might want to configure Apache to cache your files for a shorter period of time. In our case we configured Apache to cache js and css files for an hour.

The Apache documentation recommends to make configuration changes inside httpd main server config file rather that using `.htaccess` files (see http://httpd.apache.org/docs/2.4/howto/htaccess.html). Below is our caching settings:

```
ExpiresActive On
ExpiresByType image/png "now plus 1 month"
ExpiresByType image/jpeg "now plus 1 month"
ExpiresByType image/gif "now plus 1 month"
ExpiresByType application/javascript "now plus 1 hour"
ExpiresByType application/x-javascript "now plus 1 hour"
ExpiresByType text/javascript "now plus 1 hour"
ExpiresByType text/css "now plus 1 hour"
```

13.6 Notes

Due to cross-domain communication issues, the files communicating with the Django server must be hosted on a server and that server must be listed in the `XS_SHARING_ALLOWED_ORIGINS` variable in the `settings.py` file. For OpenDSA development, we host our files on `http://algoviz-beta.cc.vt.edu`.

To enable OpenDSA to communicate with the Django server, you must include the IP of your server in your book instance configuration file.

CHAPTER 14

Back-end web services and user interface

The OpenDSA "backend" API provides access to resources stored in the database. OpenDSA implements RESTful (REpresentional State Transfer) service server-side. Information is transferred to/from the back end using JSON.

14.1 Users Resources

14.1.1 Create new user

`HTTP method`: POST

`POST parameters`:

Parameters	Description
username	username of user
password	new user password
email	new user email address

`Server Response`: HTTP status[bad request], [error message].

14.1.2 Login user

`HTTP method`: POST

`POST parameters`:

Parameters	Description
username	username of user
password	new user password

`Server Response`: HTTP status[bad request|unauthorized|forbidden], [session key]

14.1.3 Logout user

`HTTP method`: POST

`POST parameters`:

Parameters	Description
key	**session key (generated by** the server)

`Server Response`: HTTP status[bad request|unauthorized]

14.2 Books Resources

14.2.1 Add new book

`HTTP method`: POST

`POST parameters`:

Parameters	Description
book_name	Name of the book
book_url	BOOK'S URL

`Server Response`: HTTP status[bad request|unauthorized]

14.3 Modules Resources

14.3.1 Add new Module

`HTTP method`: POST

`POST parameters`:

Parameters	Description
name	Name of the module
exercises	names of the exercises in the module

`Server Response`: HTTP status[bad request|unauthorized]

14.4 Exercises Resources

`HTTP method`: POST

`POST parameters`:

Parameters	Description
name	Name of the exercise
description	Description of the exercise
author	Exercise author
covers	**Topic covered by the** exercise
ex_type	Type of exercise: Khan academy, JSAV proficiency, or JSAV slide shows
streak (KA exercises)	streak of correct exercise for proficiency

`Server Response`: HTTP status[bad request|unauthorized]

14.4.1 UserModule Resources

`Endpoint`: ismoduleproficient

`HTTP method`: POST

`POST parameters`:

Parameters	Description
key	session key
module	module name

`HTTP Response`: HTTP status[bad request|unauthorized], user module proficiency status

14.5 UserExercises Resources

14.5.1 userexercise logs (KA exercises)

`HTTP method`: POST

`POST Parameters`:

Parameters	Description
key	session key
exercise	exercise name
module	module name
attempt_number	Counter for how many time the exercise has been attempted
attempt_content	Student answer
complete	1 if the answer is correct 0 otherwise
count_hints	Counter for how many time hints were use
time_taken	**Time taken to complete** the exercise
remote_adrr	IP address

`Action triggered`: Update proficiency field accordingly.

`HTTP Response`: HTTP status[bad request|unauthorized], proficiency status.

14.5.2 userexercise logs (JSAV exercises)

`HTTP method`: POST

`POST Parameters`:

Parameters	Description
key	session key
exercise	exercise name
uiid	exercise unique id
module	module name
tstamp	time the exercise was done
score	number of correct steps
total_time	**Time taken to complete** the exercise
remote_adrr	IP address

`Action triggered`: Update proficiency field accordingly.

`HTTP Response`: HTTP status[bad request|unauthorized], proficiency status.

14.6 Userinterface Resources

14.6.1 userinterface logs

`HTTP method`: POST

`POST Parameters`:

Parameters	Description
key	session key
exercise	exercise name
module	module name
book	book name
type	type of interaction
uiid	exercise unique id
desc	description of the interaction
tstamp	time the exercise was done
remote_adrr	IP address

`HTTP Response`: HTTP status[bad request|unauthorized], starage status.

14.7 Scheduled Tasks

14.7.1 Daily summary logs

We use `celery` (http://celeryproject.org/) to run scripts that create daily activity log files. The scripts run at the same time everyday. The scripts query the database and update the logs files that are used by the visualization webpage.

CHAPTER 15

Data collection server database tables

Below is the description of all the tables created by the opendsa django app models.py file.

15.1 Exercise

Stores exercises information.

Column	Type	Description
name	int	Name of the exercise
description	longtext	Description of the exercise
author	varchar	Exercise author
covers	longtext	**Topic covered by the** exercise
ex_type	varchar	Type of exercise: Khan academy, JSAV proficiency, or JSAV slide shows
streak (KA exercises)	decimal	streak of correct exercise for proficiency

15.2 Books

Stores books information.

Column	Type	Description
book_name	varchar	Name of the book
description	longtext	Book's URL

15.3 Assignments

Stores assignments information.

Column	Type	Description
course_module(1)	int	reference to assignment name
assignment_book(2)	int	reference to book
assignment_exercises	**comma separated** integers	Exercise author

`Foreign keys:` (1) Exercise_CourseModule table (A+ database), (2) Book table.

15.4 Module

Stores modules information.

Column	Type	Description
short_display_name	varchar	Name of the module
name	longtext	Description of the module
author	varchar	Exercise author
covers	longtext	**Topic covered by the** exercise
exercises_list	comma separated integers	List of exercises in the module

15.5 BookModuleExercise

Stores information related to books, modules, and exercises relationships.

Column	Type	Description
book(1)	int	Reference to book
module(2)	int	Reference to module
exercise(3)	int	Reference to exercise
points	int	Points of the exercise

`Foreign keys:` (1) Book table, (2) Module table, (3) Exercise table.

15.6 BookChapter

Stores information about chapters in a book.

Column	Type	Description
book(1)	int	Reference to book
name	int	Name of the chapter
module_list	comma separated integers	List of modules in the chapter

`Foreign keys:` (1) Book table.

15.7 UserBook

Stores book-user relationships and if user's work should be graded.

Column	Type	Description
book(1)	int	Reference to book
user(2)	int	Reference to user
grade	boolean	Indicates if user's grades should be displayed in teachers' view

`Foreign keys:` (1) Book table, (2) Auth_User table (Django table).

15.8 UserButton

Stores clickstream/interactions data

Column	Type	Description
book(1)	int	Reference to book
user(2)	int	Reference to user
exercise(3)	int	reference to exercise
module(4)	int	reference to module
name	varchar	type of interaction
description	longtext	description of the interaction
action_time	datetime	time of the interaction
uiid	int	unique id of the intercation
browser_family	varchar	browser used by the user
browser_version	varchar	browser's version
os_family	varchar	operating system used
os_version	varchar	operating system's version
device	varchar	device used by the user
ip_address	varchar	IP address of the user

`Foreign keys:` (1) Book table, (2) Auth_User table (Django table), (3) Exercise table, (4) Module table.

`Jsav buttons` jsav-forward: go to the next slide. jsav-backward: back to the previous slide. jsav-begin: go to the first page of the slideshow. jsav-end: go to the last page of the slideshow. => With AV information (see below), those actions would be very useful to calculate which slides are most viewed, and it would give a different aspect to calculate the slide reading time.

`Mouse focus` window-focus: student is looking at this page. window-blur: student left this page. => With this information, we might be able to calculate students' actual spent time on OpenDSA.

`Module load` window-unload: in the current system, this actions is recorded only when students leave the page, but within Canvas, it is recorded when users hit the next button to continue reading the prose. document-ready: a module is loaded.

`KA exercise` load-ka: this action is recorded when KA exercise framework is loaded. When KA exercise framework is loaded, all interaction logs go to userexerciselog, so userbutton table does not get any interaction log. However, when the KA exercise is refreshed (for gaming or any other reason), this actin is recorded on the userbutton table. Therefore, by counting the frequency of this action, we can tell how many times students refreshed the page to avoid hard questions. With a new infrastructure, we are getting an exact exercise name, so along with these two information, we can catch one type of gaming activity with confidence.

`AV information` ev_num: number of clicks on Jsav with any jsav button (forward, backward, begin, or end). currentStep: number of required clicks to reach the last slide. Thus, the total number of slide is the value of currentStep + 1. (e.g. if there is a set of slide with five slides, the currentStep value is four since you need to click four times to get to the end of the slide show).

15.9 UserModule

Records a summary of a student activity on a module

Column	Type	Description
book(1)	int	Reference to book
user(2)	int	Reference to user
module(3)	int	reference to module
first_done	datetime	date of first module attempt
last_done	datetime	date of last module attempt
proficient_date	datetime	date of prociciency

`Foreign keys:` (1) Book table, (2) Auth_User table (Django table), (3) Module table.

15.10 UserData

Records summary of user/exercises activity (started exercises and proficient exercises).

Column	Type	Description
book(1)	int	Reference to book
user(2)	int	Reference to user
started_exercises	comma separated int	reference to module
proficient_exercises	comma separated int	date of first module attempt

`Foreign keys:` (1) Book table, (2) Auth_User table (Django table)

15.11 UserExerciseLog

Records information about each exercise attempt.

Column	Type	Description
user(1)	int	Reference to user
exercise(2)	int	reference to exercise
correct	boolean	correctness of the attempt
time_done	datetime	time of the attempt
time_taken	datetime	time taken to complete the exercise
counts_hints	int	number of hint used
hint_used	boolean	hint used or not
earned_proficiency	boolean	proficiency earned or not
count_attempt	int	number of attempts
ex_question	varchar	KA question name
request_type	longtext	KA user interaction type

`Foreign keys:` (1) Book table, (2) Exercise.

15.12 UserProgLog

Records additional information about each programing exercise attempt.

Column	Type	Description
problem_log(1)	int	Reference to UserExerciseLog
student_code	longtext	students attempt code
feedback	longtext	compilation feedback

`Foreign keys:` (1) UserExerciseLog table.

15.13 UserProfExerciseLog

Stores additional information about each JSAV proficiency exercise attempt.

Column	Type	Description
problem_log(1)	int	Reference to UserExerciseLog
student	int	
correct	int	
fix	int	
undo	int	
total	int	

`Foreign keys:` (1) UserExerciseLog table.

15.14 UserExercise

Stores statistics about student attempts each record contains the data related to one student and one exercise.

Column	Type	Description
user(1)	int	Reference to user
exercise(2)	int	reference to exercise
streak	int	streak of the attempt (KA)
longuest_streak	int	user's longuest streak
first_done	datetime	time of first attempt
last_done	datetime	time of last attempt
total_done	int	total number of attempts
total_correct	int	number of correct attempts
proficient_date	datetime	date of proficiency
progress	decimal	correctness percentage
correct_exercises	longtext	a list of correct exercises
current_exercises	longtext	current exercise title
hinted_exercises	longtext	recently hint used exercise

`Foreign keys:` (1) Auth_User table (Django table), (2) Exercise table.

15.15 Bugs

Stores bugs reported by users.

Column	Type	Description
user(1)	int	Reference to user
os_family	varchar	OS where the bug occured
browser_family	varchar	browser used
title	varchar	short summary of the bug
description	longtext	detailed bug's description
screenshot	longtext	path to the screenshot

`Foreign keys:` (1) Auth_User table (Django table).

CHAPTER 16

Administrator's Tools

16.1 Teacher View Configuration

To configure a view for a specific course/book instance, it is required to create a course and an assignement for that course through the backend administration console. Only the Django administration account can perform these tasks.

16.2 Course Creation

If this is a course that does not exist, then on the A+ homepage under `Courses`, click on `+Add`, fill in the form, and assign teacher(s) to the course.

On the A+ homepage under `Course instances` click on `+Add` to create an instance of the course. Fill in the form and assign teaching assistants.

On the A+ homepage under `Bookss`, you should now see your book's URL listed if the book has been compiled. Click on the url of the OpenDSA book instance you want to associate to the course. In the book's form, select the appropriate course to associate it to.

16.3 Granting Instructor Access

To give instructor access to an existing account, from the django administration page go to "Courses" to display the list of available courses, then click on the course ID to edit course settings.

16.4 Register a Book Instance

To send over to the data server all information (chapters, modules, exercises, points, etc.) related to a newly created book instance, it is necessary to register the book instance. The registration can be done on the `RegisterBook` page at `Book_Base_URL/html/RegisterBook.html`. For example, `http://algoviz.org/OpenDSA/Books/OpenDSA/html/RegisterBook.html`. This presumes that the book instance has already been configured by a Django administrator.

16.5 Managing the Front Page

The “front page” for the instructor’s view will list a separate button for each “active” course. An “active” course is any course who’s end date is past the current moment in time. All courses will be listed on the “course archive” page.

CHAPTER 17

Client-Side Framework

The client-side framework for OpenDSA is implemented in 3 main files located in the lib directory: `odsaMOD.js`, `odsaAV.js` and `odsaUtils.js`. `odsaMOD.js` contains the code which is specific and common to modules, while `odsaAV.js` contains code specific and common to AVs. `odsaUtils.js` contains several utility functions as well as the interaction data logging functions used by both `odsaMOD.js` and `odsaAV.js`. AVs must include both `odsaUtils.js` and `odsaAV.js` (in that order) to function properly as part of OpenDSA. While the script links for modules are automatically appended during the configuration process, modules must include `_static/config.js`, `odsaUtils.js` and `odsaMOD.js` (in that order). `config.js` is a file generated by the configuration process that stores configurable settings needed by the client-side framework. **Note**: AVs should not include `odsaMOD.js` and modules should not include `odsaAV.js`.

Be aware that modules and AVs actually load the minimized version of these files (such as `odsaAV-min.js`). So if you edit one of the library files, be sure to run:

```
make min
```

from the OpenDSA toplevel before you check to see the effect.

In order to allow AVs to be reused outside of OpenDSA (without including OpenDSA infrastructure), `odsaAV.js` checks for all dependencies provided by `odsaUtils.js` and provides default values or function stubs to maintain its independence. Additionally, AVs should not be made dependent on anything generated by the configuration process (such as `config.js`).

The client-side framework is responsible for:

- Allowing users to login, logout, and register new accounts
- Dynamically resizing the iFrames for embedded exercises
- Sending the information necessary to store a new exercise in the database
- Managing a user's score
  - Automatically buffering and sending score data to the backend server when a user completes an exercise
- Managing a user's proficiency
  - Determining whether a user obtains proficiency with a module or exercise
  - Caching the user's proficiency status locally
  - Displaying the appropriate proficiency indicators
  - Making sure the local proficiency cache remains in sync with the server
- Keeping multiple OpenDSA pages in sync

  - Ensure that actions such as logging in, logging out or gaining proficiency are reflected across all OpenDSA pages open within the browser

- Collecting and transmitting user interaction data to the backend server
- Dynamically configuring the natural language, code language, and default parameters of AVs at runtime

All but the last two of these responsibilities are handled by `odsaMOD.js`, effectively making module pages the "brains" of the client-side framework. We wanted the client to be able to determine whether or not a user obtained proficiency with an exercise so that OpenDSA could function without a backend server, but in order to do so, the client needs to know the exercise's threshold value. The threshold is configurable and we didn't want to compile AVs because it would raise the barrier of entry for AV development (which currently only requires a text editor and a browser), so the solution was to include the threshold (and other exercise-specific information) on the module page when it was built. The next challenge was to get the score from embedded AVs to the module page which was easily implemented using HTML5 postMessage. While the current configuration system could send the threshold to embedded AVs as a URL parameter, we choose to leave the system the way it is because it makes sense to only grade an exercise within a specific context. Exercises don't intrinsically have points or a proficiency threshold. These are properties of the context in which the exercise is viewed (i.e. they are book dependent). Therefore it makes sense to have exercises graded by code on the module page and to have module pages handle submitting a user's score and displaying an indicator of their proficiency.

Each of the main JavaScript files are wrapped in anonymous functions to hide their internal variables and functions. Public functions are accessible through the global ODSA object. Global settings can be accessed through ODSA.SETTINGS, public utility functions can be accessed through ODSA.UTILS, public module functions through ODSA.MOD and public AV functions through ODSA.AV.

17.1 Responsibilities

17.1.1 Login, Logout and Registration

Unlike AVs and Khan-Academy exercises, module pages include links to login, logout and register a new account. The HTML for the login and registration boxes can be found in `~OpenDSA/RST/source/_themes/haiku/layout.html` and, of course, the code for it can be found (clearly marked) in `~OpenDSA/lib/odsaMOD.js`. `showRegistrationBox()` and `showLoginBox()` make the respective pop-up boxes appear and ensure they are centered correctly on the page, while `hidePopupBox()` hides which ever pop-up box is showing. `login(username, password)` sends an AJAX request to the server with the provided username and password and if successful will create a new session for the given user. When the page loads, click handlers are attached to the various login and registration links and buttons to trigger the appropriate behavior. Clicking the "Login" or "Register" links will open the appropriate pop-up box, clicking "Submit" in the login box will trigger the `login(username, password)`, and clicking "Submit" in the registration box will cause the user's input to be validation and if successful an AJAX message is sent to the server requesting a new user account. If the user's input is invalid an error message is displayed and if registration is successful `login(username, password)` is called to automatically log the user in.

Since the database only handles one session per user at a time, if the user logs in anywhere else, their first session is invalidated. If the user triggers any communication to the server using the old, invalid session key, the server will return an HTTP 401 error causing the framework to call `handleExpiredSession(key)`. This function removes the old session information from local storage, informs the user that their session is no longer valid and that they must log in again, then refreshes the page when the user closes the message which causes the login box to pop up. The module page also implements a listener for "odsa-session-expired" events which are generated by the event logging functions in `odsaUtils.js` if they receive an HTTP 401 error. When these events are received they call `handleExpiredSession(key)`.

If no backend server is enabled, the login and registration links are hidden because they serve no function if there is no server to log into.

17.1.2 Dynamic iFrame Resizing for Embedded Exercises

The client-side framework supports changing the height and width of embedded exercise iFrames at runtime. `odsaAV.js` sends an HTML5 postMessage to the parent module page when the AV loads or is reset, communicating the height and width of the rendered page. A listener defined in `odsaMOD.js` receives the message and updates the dimensions of the iFrame associated with the exercise and hiding the iFrame, if applicable. **Note**: If the iFrame is hidden when the exercise is loaded, the dimensions may not be reported properly, so the iFrame must be hidden after it has been loaded and resized.

Due to the way Khan-Academy exercises can contain multiple problems of different sizes, the overall exercise must use data attributes of the exercise's body element to define the largest necessary height and width, as seen in the following example:

```
<body data-height="650" data-width="950">
  <div class="exercise" data-name="ExchangeTF1"></div>
...
```

These data attributes are read from the Khan-Academy exercise file by the `avembed` directive during the compilation process and used to set the dimensions of the exercise's iFrame.

17.1.3 Dynamically Loading Exercises

One advantage to having all the configuration information for modules and exercises available on the client is that it provides an easy way to load exercises into the database that do not already appear there. A function called `loadModule()` is called when a page loads which handles several conditions. If a user is logged in, it sends an AJAX request to the server which contains enough information to load the module and all the exercises it contains if they do not already exist in the database. The response from the server contains information about the user's proficiency with the module, each exercise in the module and progress information for the Khan Academy-style exercises. The local proficiency cache is updated based on the information in the response which keeps the client in sync with the server. If no user is logged in when `loadModule()` is called, the anonymous (guest) user information stored in the local proficiency cache is used to initialize the proficiency indicators on the module page.

17.1.4 Score Management

Module pages contain 3 listeners. One listens for "jsav-log-event" events which are generated by the JSAV-based mini-slideshows that are included on most module pages, while a second listens for HTML5 postMessages from embedded AVs or Khan Academy exercises. The third is not relevant to this section and is described above (see Login, Logout and Registration). The first two listeners call `processEventData(data)` which performs some processing to make sure all additional event data is logged properly and calls `storeExerciseScore(exercise, score, totalTime)` under 3 circumstances: if the event type is "odsa-award-credit", if the user has reached the end of a slideshow (and all the steps were viewed or the book is configured to allow credit without viewing all the slides), and if the event type is "jsav-exercise-grade-change" and the final step in the exercise was just completed. If a user is logged in or the system is configured to assign anonymous score data to the next user who logs in `storeExerciseScore()` will create a score object and store it in local storage in accordance with the Score Data model below. If the score is above the proficiency threshold and either no backend server is enabled or no user is logged in, the anonymous (guest) user is awarded proficiency and the appropriate proficiency indicator is displayed.

JSAV does not communicate directly with the OpenDSA backend and does not tell the backend to award credit for an exercise. In the case of proficiency exercises, JSAV generates a "jsav-exercise-grade-change" event which contains the student's points and the total number of points for the exercise. The OpenDSA client-side framework calculates the student's score and compares it to the threshold value for the exercise that was provided in the configuration file. If the score is greater than or equal to the threshold, credit is awarded locally. The calculated score is packaged up and sent to the backend which makes an independent comparison to the threshold that has previously been sent by the client and verifies whether or not the student should obtain credit.

Some OpenDSA functions such as `awardCompletionCredit()` and `logExerciseInit()` generate events on the same channel as JSAV in order to make use of the existing listener. While they communicate on the same channel, these functions are not associated with JSAV and are NOT dependent on the JSAV framework.

Near the end of `processEventData()`, `flushStoredData()` is called which in turn calls `sendExerciseScores()` and `sendEventData()` (which is defined in `odsaUtils.js`). `sendExerciseScores()` loops through local storage calling `sendExerciseScore()` for any score events with a timestamp less than the timestamp taken when the function was called. `sendExerciseScore()` sends the specified score object to the backend server and updates the user's proficiency status for the exercise based on the server's response. If the score was sent successfully or was rejected by the server, the object is removed from local storage. In the case of rejection, the data is removed to prevent a build up of bad data that will never succeed and be cleared. If transmission is unsuccessful for another reason, the score object will remain in local storage and the framework will attempt to send it again in the future.

17.1.5 Proficiency Management

The module page is also in charge of determining a user's proficiency with an exercise or module, caching this proficiency status in local storage, displaying the appropriate proficiency indicator for each exercise and making sure the local proficiency cache stays in sync with the server. For each book, for each user, the client stores the status of each exercise with which the user obtains proficiency. The status can be one of several states:

- **SUBMITTED** - indicates the user has obtained local proficiency and their score has been sent to the server
- **STORED** - indicates the user has obtained local proficiency and the server has successfully stored it
- **ERROR** - indicates the user has obtained local proficiency, the score was sent to the server but it was not stored successfully
- If an exercise does not appear in a user's proficiency cache, that user has not obtained proficiency

Local Proficiency Cache

The primary purpose of the local proficiency cache is to allow anonymous (guest) users to maintain their progress and to allow OpenDSA to function without a backend server, but a secondary purpose is to make pages more responsive for logged in users. While `loadModule()` (which is called on every page when a user is logged in) returns the user's proficiency information, keeping a local copy allows the page to immediately display the proper proficiency indicators rather than waiting for a response from the server. See Proficiency Data for information about the format of the cached data.

Proficiency Displays

Proficiency for mini-slideshows is indicated by the appearance of a green checkmark on the right side of the slideshow container. If the status is `SUBMITTED`, a "Saving..." message will appear beneath the checkmark but will be hidden once the status changes to `STORED`. If the status is set to `ERROR`, a warning indicator will appear (to draw the user's attention to the exercise) and the saving message will be replaced by an error message and a "Resubmit" link which allows the user to resend their score data without recompleting the exercise.

Proficiency for embedded exercises is indicated by the color of the button used to show or hide the exercise. Red indicates the user is not proficient, yellow indicates the user's score has been submitted or an error occurred and green indicates that the user is proficient (and their proficiency has been verified by the server).

When a user obtains proficiency for all the required exercises in a module, the words "Module Complete" will appear in green at the top of the module. If "Module Complete" appears in yellow, the user has obtained local proficiency with all the required exercises but one or more of them have not yet been successfully verified by the server (this should ONLY appear when a user is logged in). In general, to obtain module completion a user must complete all exercises marked as "required" in the configuration file. If a module does not contain any required exercises, module completion cannot be obtained unless the configuration file sets "dispModComp" to "true" for the given module. Inversely, if "dispModComp" is set to "false" module completion will not be awarded even if the user completes all the required exercises.

On the Contents (index) page, a small green checkmark next to a module indicates that it is complete.

On the Gradebook page, the score for exercises and modules with which the user is proficient are highlighted in green. At this time, there is no concept of chapter completion.

All updates to proficiency displays are handled by `updateProfDisplay()`. Code within the function determines what displays exist for the given exercise or module and updates them according to the associated status stored in the local proficiency cache.

Syncing with the Server

As described above, under Dynamically Loading Exercises, `loadModule()` is called when each module page loads and the response contains information about the user's proficiency with the module and each exercise in the module.

The Contents (index) and Gradebook pages call `syncProficiency()` which initiates an AJAX request to the backend server which in turn responds with the proficiency for all modules and exercises.

In both cases, the information returned by the server is used to update the local proficiency cache.

Determining Proficiency Status

Proficiency status is determined differently in different situations. If no backend server is enabled or no user is logged in (meaning the user is anonymous / guest), the client is given the authority to determine whether or not a user is proficient with an exercise or module. Exercise proficiency is awarded if the user's score on an exercise is greater than or equal to the proficiency threshold for that exercise. Module proficiency is awarded when a user has obtained proficiency with all exercises in a module that are listed as "required" in the configuration file. Since there is no server involved in the process, the only valid status for anonymous (guest) users is `STORED`.

The backend server is required to verify proficiency of all logged in users and two additional statuses are added to handle interaction with the server. When a logged in user's exercise score is sent to the server, if the client determines they are proficient, their status for the given exercise is set to `SUBMITTED`. When the server responds to the AJAX request, the response contains a boolean indicating whether or not the user is proficient with the given exercise. If the server determines the user is proficient, their status for the exercise is set to `STORED`, but if the server responds with `"success": false` or an HTTP error occurs, the status is set to `ERROR`.

When the status of a required exercise is set to `STORED` (in `storeStatusAndUpdateDisplays()`), the framework calls `checkProficiency(moduleName)` to check for module proficiency. `checkProficiency()` begins by calling `updateProfDisplay()` which updates the proficiency displays for the given exercise or module based on the contents of the local proficiency cache and returns the status. If the status is `STORED`, `checkProficiency()` returns immediately. If the status is not `STORED` but a user is logged in, the framework will send an AJAX request to the backend server asking if the user is proficient with the exercise or module and update the proficiency cache appropriately when it receives a response. If the status is not `STORED`, no user is logged in and the request is for module proficiency, `checkProficiency()` will loop through the `exercises` object (see Exercises) and determine if the anonymous (guest) user has proficiency with all required exercises. If so, the guest account is awarded module proficiency and the cache is updated. If a single required exercise is found that the guest user is not proficient with, the loop short circuits and the function returns.

A user's proficiency status can also be updated by the synchronization functions `loadModule()` and `syncProficiency()` (see Syncing with the Server).

17.1.6 Keeping Pages in Sync

Consider the situation where a user logs in to OpenDSA and then opens modules in multiple tabs. Since a user is logged in each tab will display the logged in user's name in the top right hand corner. Later, the user logs out and another user logs in on one of the pages. Without a system to sync pages, it would appear as if two users are logged in at the same time which could potentially be very confusing. To rectify this situation, `odsaMOD.js` implements an `updateLogin()` function which is called any time the window receives focus. The purpose of this function is to determine whether or not the current user appears to be logged in and if not to fix it. If another user has logged in since the page was loaded, the former user's name is replaced with the current user's name and if no user is logged in, the logout link and former user's name are replaced with the default "Register" and "Login" links. If any change is made, `loadModule()` is called to ensure the proficiency displays match the current user's progress. Since the function is called when the window receives focus, updates will be made as soon as the user clicks on the tab to open it.

17.1.7 Interaction Data Collection and Transmission

We collect data about how users interact with OpenDSA for two reasons

1. To continually improve OpenDSA
2. For research purposes

As a user interacts with OpenDSA, a variety of events are generated. If there is a backend server enabled, we record information about these events, buffering it in local storage and sending it to the server when a flush is triggered. If a user is logged in, we send the event data with their session key, effectively tying interaction data to a specific user, but if no user is logged in the data is sent anonymously (using 'phantom-key' as the session key). This ensures that we are able to collect as much interaction data as possible.

17.1.8 Runtime Exercise Configuration Support

The client-side framework supports limited dynamic configuration at runtime through the use of JSON exercise configuration files (not related to the JSON config file used by the configuration system). Configuration currently supports:

- Natural language switching
- Code language switching
- Default parameter configuration

While the natural language of module text and code language of code snippets are set by the configuration system when the book is built (by the configuration system and codeinclude directive, respectively), exercises and some interface elements are (intentionally) not altered by the configuration process and therefore must be configured at runtime. We take extreme measures to keep from having to alter the exercises during the configuration process so that we can load the same exercise on different book instances and to lower the barrier of entry for new AV developers so that the only tools they need are a text editor and a browser rather than our entire tool chain.

The function responsible for this is `loadConfig()` in `odsaUtils.js`. It uses AJAX to load the appropriate JSON exercise configuration file, does additional processing and loading as needed to obtain the natural language translations, applies translated labels to interface elements, loads the appropriate code snippet if it exists, and applies the default parameters from the configuration file to the `PARAMS` object such that any conflicting parameters are overridden unless the parameter is set via the URL.

JSON File Locations

The framework assumes that standalone AVs and mini-slideshows follow the convention of having a config file [av_name].json in the same directory as the JS file the defines the AV, if the path to the JSON file is different (in a different directory, a common JSON file is shared between AVs, etc), the path relative to the OpenDSA root directory must be specified using the "json_path" argument. Example: `ODSA.UTILS.loadConfig({"json_path": "AV/Sorting/shellsortAV.json"});`

The `av_name` argument defaults to `ODSA.SETTINGS.AV_NAME` which should work out of the box for all standalone AVs, but is not initialized on modules pages, making this argument required for mini-slideshows.

By convention the ID of the container containing the AV defaults to `#container` for standalone AVs and `#[av_name]` (auto-generated by the inlineav directive) for mini-slideshows, as long as you follow this convention, you should not have to provide this argument

JSON Format

The JSON exercise configuration file may contain the keys: `translations`, `code`, and `params`. Each key under `translations` should be an ISO-639 standardized language code. For keys beneath a language code key, if the key is prefixed with `av_` it will be ignored by the framework and left up to the AV developer to explicitly reference it. All other keys will be evaluated as a jQuery selector and the associated string applied to the element returned.

Each key under `code` corresponds to a programming language which must have a matching folder in the SourceCode/ directory. Note that while the directory in SourceCode/ may contain capitals, the key must be all lowercase. This standard was adopted to ensure consistent key names across AV authors (i.e. prevent one author from using `Java` while another uses `java`, etc)

```
{
  "translations" : {
    "en": {
      ".avTitle": "Insertion Sort Visualization",
      "av_Authors": "Cliff Shaffer and Nayef Copty",
      "#about": "About",
      "#run": "Run",
      "#reset": "Reset",
      "#arraysizeLabel": " List size: ",
      "#arrayValuesLabel": " Your values: ",
      "av_arrValsPlaceholder": "Type some array values, or click 'run' to use random values",
      "av_c1": "Starting Insertion Sort.",
      "av_c2": "Done sorting!",
      "av_c3": "Highlighted yellow records to the left are always sorted. We begin with the record i
      "av_c4": "Processing record in position ",
      "av_c5": "Move the blue record to the left until it reaches the correct position.",
      "av_c6": "Swap."
    },
    "fi": {
      ".avTitle": "Lomitusjärjestäminen",
      "av_Authors": "Cliff Shaffer and Nayef Copty",
      "#about": "Lisätietoa",
      "#run": "Suorita",
      "#reset": "Alusta",
      "#arraysizeLabel": " Taulukon koko: ",
      "#arrayValuesLabel": " Omat arvot: ",
      "av_arrValsPlaceholder": "Erottele arvot välilyönnillä tai jätä tyhjäksi satunnaislukuja varten
      "av_c1": "FIStarting Insertion Sort.",
      "av_c2": "FIDone sorting!",
      "av_c3": "FIHighlighted yellow records to the left are always sorted. We begin with the record
      "av_c4": "FIProcessing record in position ",
      "av_c5": "FIMove the blue record to the left until it reaches the correct position.",
      "av_c6": "FISwap."
    },
    "sv": {
      ".avTitle": "Visualisering av Mergesort",
      "av_Authors": "Cliff Shaffer and Nayef Copty",
      "#about": "Om",
      "#run": "Kör",
      "#reset": "Återställ",
      "#arraysizeLabel": " Liststorlek: ",
      "#arrayValuesLabel": " Dina värden: ",
      "av_arrValsPlaceholder": "Skriv in dina värden eller lämna blankt för att använda slumpmässiga
      "av_c1": "SVStarting Insertion Sort.",
      "av_c2": "SVDone sorting!",
      "av_c3": "SVHighlighted yellow records to the left are always sorted. We begin with the record
      "av_c4": "SVProcessing record in position ",
      "av_c5": "SVMove the blue record to the left until it reaches the correct position.",
      "av_c6": "SVSwap."
    }
  },
  "code" : {
    "processing": {
      "url": "../../SourceCode/Processing/Sorting/Insertionsort.pde",
      "startAfter": "/* *** ODSATag: Insertionsort *** */",
      "endBefore": "/* *** ODSAendTag: Insertionsort *** */",
      "lineNumbers": false,
      "tags": {
        "sig": 1,
        "outloop": 2,
        "inloop": 3,
        "swap": 4,
        "end": 5
      }
    },
    "c++": {
      "url": "../../SourceCode/C++/Sorting/Insertionsort.cpp",
      "startAfter": "/* *** ODSATag: Insertionsort *** */",
      "endBefore": "/* *** ODSAendTag: Insertionsort *** */",
      "tags": {
        "sig": 1,
        "outloop": 2,
        "inloop": 3,
        "swap": 4,
        "end": 5
      }
    },
    "java": {
      "url": "../../SourceCode/Java/Sorting/Insertionsort.java",
      "startAfter": "/* *** ODSATag: Insertionsort *** */",
      "endBefore": "/* *** ODSAendTag: Insertionsort *** */",
      "tags": {
        "sig": 1,
        "outloop": 2,
        "inloop": 3,
        "swap": 4,
        "end": 5
      }
    }
  },
  "params": {
    "JXOP-lang": "en"
  }
}
```

Control

Control over the natural language of an exercise is done by setting either `JSAV_EXERCISE_OPTIONS.lang` or `JSAV_OPTIONS.lang`, while the code language is controlled by `JSAV_EXERCISE_OPTIONS.code` or `JSAV_OPTIONS.code`.

For embedded AVs, these options can be set several different ways:

1. Hardcoding a setting into the framework itself (rare)
2. Using the `params` field of the exercise configuration file.
3. Using the `glob_exer_options` field in the book configuration file.
4. Using the `exer_options` field related to a specific exercise in the book configuration file.

Method 1 Example

```
JSAV_EXERCISE_OPTIONS.lang = 'en';
JSAV_EXERCISE_OPTIONS.code = 'c++';
```

Method 2 Example

```
{
  ...,
  "params": {
    "JXOP-lang": "en",
    "JXOP-code": "c++"
  }
}
```

Method 3 Example

```
{
  ...,
  "glob_exer_options": {
    "JXOP-lang": "en",
    "JXOP-code": "c++"
  },
  ...,
}
```

Method 4 Example

```
{
  ...,
  "chapters": {
    ...,
    "Algorithm Analysis": {
      ...,
      "AlgAnal/AnalProgram": {
        ...,
        "exercises": {
          "binarySearchCON": {
            "exer_options": { "JXOP-code": "none" },
            ...
          },
        }
      },
      ...,
    },
    ...,
  }
}
```

For mini-slideshows, the first two methods from above apply, but options three and four use `glob_mod_options` and `mod_options`, respectively. See *Configuration* for more information.

The order of precedence is such that the later methods will override the previous ones. If the preferred natural language is not present in the configuration file, the framework will default to English. If the preferred code language is not present, the framework will default to the first code language defined in the file. If the code language is set to `none` or the code object is entirely omitted from the config file, then code display will be disabled for the AV.

17.2 Data Model

The following sections describe the format of different data structures used for the client-side framework.

17.2.1 Exercises

Each module page creates an `exercises` object on page load which is used to quickly and easily access important information about the module's exercises. Each exercise object in `exercises` includes:

- Points - the number of points the exercise is worth
- Required - whether or not the exercise is required for module proficiency
- Threshold - the minimum score a user must receive to obtain proficiency
- Type - the type of exercise
  - 'ka' for Khan Academy style exercises
  - 'pe' for proficiency exercises
  - 'ss' for slideshows
- uiid (unique instance identifier) - a code that uniquely identifies an instance of an exercise, used to group log events

Example of `exercises`

```
{
  "shellsortCON1": {
    "points": 0.1,
    "required": true,
    "threshold": 1.0,
    "type": ss,
    "uiid": 1362467525562
  },
  "ShellsortProficiency": {
    "points": 1.1,
    "required": true,
    "threshold": 0.9,
    "type": pe,
    "uiid": 1362467577655
  }
}
```

17.2.2 Score Data

- When a user completes an exercise, a score object is generated and saved to local storage using a key of the form: 'score-[timestamp]-[random_number]'. The 'score' prefix identifies the object as score data, while the timestamp helps make the key unique and allows the framework to quickly determine whether the associated score data was recorded prior to the timestamp taken when the send function was called. The random number ensures that if two events are logged at the exact same time they will not overwrite each other. While possible, the probability of two events being logged at the exact same time and having the same random number is negligible.
- The OpenDSA framework will attempt to send the score to the server immediately. If the score was sent successfully or was rejected by the server, the object is removed from local storage. In the case of rejection, the data is removed to prevent a build up of bad data that will never succeed and be cleared. If transmission is unsuccessful for another reason, the score object will remain in local storage and the framework will attempt to send it again in the future or when the user clicks the 'Resubmit' button associated with an exercise.
- If no user is logged in, score data will still be cached, but not sent to the server. When a user logs in, all anonymous score data is awarded to that user (if OpenDSA is configured to do so).
- Each score data object contains the following fields:
  - exercise - the name of the exercise with which the score is associated
  - book - the identifier of the book with which the event is associated
  - module - the module the event is associated with
  - score - the user's score for the exercise
  - steps_fixed - the number of steps fixed during the exercise
  - submit_time - the timestamp of when the user finished the exercise
  - total_time - the total amount of time the user spent working on the exercise
  - uiid - the unique instance identifier which allows an event to be tied to a specific instance of an exercise or a specific load of a module page
  - username - the username of the user who earned the score

Example:

```
localStorage["score-1377743343193-24"] = '{"exercise":"SelsortCON1","module":"SelectionSort","score"
```

17.2.3 Proficiency Data

The proficiency status of each completed exercise and module is stored in local storage using a key of the form: 'prof-[username]-[bookID]-[module_or_exercise_name]'. The prefix 'prof' identifies the data as cached proficiency data, while the username, bookID, and module / exercise name identifies what the status is related to.

Example:

```
localStorage["prof-testuser-d8a4a0d9967b6722a417e411bf13f9b99005c851-IntroDSA"] = "STORED"
```

17.2.4 Interaction / Event Data

- User interaction data is stored in local storage as an object with the following fields:
  - av - the name of the exercise with which the event is associated ("" if it is a module-level event)
  - book - the identifier of the book with which the event is associated
  - desc - a stringified JSON object containing additional event-specific information
  - module - the module the event is associated with
  - tstamp - a timestamp when the event occurred
  - type - the type of event
  - uiid - the unique instance identifier which allows an event to be tied to a specific instance of an exercise or a specific load of a module page
  - user - the username of the user who generated the event

Event data is stored using a key of the form: 'event-[timestamp]-[random_number]'. The 'event' prefix identifies the object as user interaction data, while the timestamp helps make the key unique and allows the framework to quickly determine whether the associated event occurred prior to the timestamp taken when the send function was called. The random number ensures that if two events are logged at the exact same time they will not overwrite each other. While possible, the probability of two events being logged at the exact same time and having the same random number is negligible.

Example:

```
localStorage["event-1377743343193-8"] = '{"type":"document-ready","desc":"{\"msg\":\"User loaded the
```

17.3 Implementation and Operation

With the exception of login, all data is sent to the server with a session key rather than the username. The server is able to recover the username from the session and this should prevent data from inappropriately being sent as a different user. Since anonymous users do not have sessions, their interaction data is sent using the hardcoded value, "phantom-key", as the session key.

17.3.1 Data Flow

As a user interacts with an AV, it generates events. A listener in `odsaAV.js` processes the events (logging additional event data in desc field, triggering certain AV specific events like displaying a message saying no credit will be given after viewing the model answer, etc), logs them and forwards the event to the parent page. The parent page may or may not implement an event listener and process them further (a flag is set to indicate the event has already been logged, to prevent duplicate logging). The module page implements such a listener and passes events from embedded pages and events generated by the module itself to `processEventData()`. Here events which have not been logged are logged and certain events trigger saving a user's score (namely moving forward to the last slide of a slideshow, completing a graded exercise, `odsa-award-credit` event used to award completion credit). In these cases, `storeExerciseScore()` is called to store the user's score in localStorage with additional information about the exercise. At the end of `processEventData()`, score and event data are pushed to the server, if necessary, using `flushStoredData()` (which calls `sendEventData()` and `sendExercisesScores()`).

17.3.2 Page Initialization

- `updateLogin()` is called on page load or when the page gains focus and functions to ensure consistency between all OpenDSA pages, specifically making sure the current user appears logged in and the proficiency indicators display that user's proficiency. Without this function, a user could log in to multiple tabs, then log out of one and still appear to be logged into the others or another user could log in and it would appear that two users were logged in on the same browser at the same time, even though all data would be submitted as the last user to log in. `updateLogin()` synchronizes all the pages to prevent confusing situations.
- `loadModule()` is called when the page loads and when `updateLogin()` updates a page to reflect a new user being logged in and performs different actions in different contexts. If the user is on the index page, `loadModule()` loops through all the linked module pages and calls checkProficiency() for each. If the user is viewing a module page, one of two things happens. If the backend server is enabled and a user is logged in, a message will be sent to the server containing all the information necessary to load the module and all exercises if they don't already appear in the database and the response from the server will contain the user's proficiency status which each exercise and the module itself (the progress is also returned which allows the client to update the progress bar on Khan Academy exercises). If no backend server is enabled or no user is logged in, `loadModule()` updates the proficiency indicators based on the anonymous user's data in local storage.

17.3.3 Support Functions

`storeStatusAndUpdateDisplays()` calls `storeProficiencyStatus()` to store the given status in the local storage, then updates the appropriate proficiency display (whether its for an exercise or a module) and checks whether or not the user is now proficient with the module (if the user just gained proficiency with an exercise)

- `storeProficiencyStatus(name, [status], [username])` takes an exercise or module name, a status (optional) and username (optional) and caches the given status for the given exercise / module for the given user in local storage. If username is not specified, the current user's name is used and if status is not specified, it defaults to `STORED`.
- `updateProfDisplay(name)` can be called with either an exercise or module name as an argument (if no argument is given, it will default to the current module name). The function automatically detects whether the argument is an exercise or module name and updates the appropriate display(s) based on the current user's proficiency status in local storage.
- `checkProficiency(name)` can be called with either an exercise or module name as an argument (if no argument is given, it will default to the current module name). This function checks local storage for the given exercise / module and if it's found, calls `updateProfDisplay()` and returns. If the exercise / module is not found, the server is queried for the user's proficiency status and when the response is received, `storeStatusAndUpdateDisplays()` is called to make sure the status is stored in local storage and the proficiency indicators are updated.

17.4 Debugging

The client-side framework is a relatively complex system which can be difficult to fully understand without tracing its execution. While the debugging tool built into Firebug can be useful for this, its impossible to back up and see something execute again or compare how a value changes without manually remembering the previous value. The current solution is to wrap console logging statements with a conditional based on the flag `localStorage.DEBUG_MODE`. To enable DEBUG_MODE simply run `localStorage.DEBUG_MODE = 'true'` from the JavaScript console. The log statements are grouped by function and internal calls are nested to make it easy to trace the call chain. Groups can be collapsed to hide information the user is not interested in and make the interesting information stand out more. It also provides a quick and easy way for a developer to scan through the log and make sure all the functions they expect to be called are called without having to step through all of them with the debugger. To disable verbose logging, run: `delete localStorage.DEBUG_MODE` from the JavaScript console.

Unfortunately, this debugging system makes the code a little more bulky and less readable, but it has been found to be very helpful for debugging. Additionally, if students are experiencing problems, this system will allow us to quickly and easily diagnose their problem on their own computer without requiring them to install Firebug or adding additional print statements to the framework itself.

17.5 MathJax Support

We use MathJax extensively to create mathematical expressions. It gets used in module text, and within AVs and Exercises. Proper use of MathJax involves providing it with necessary configuration, in addition to loading the necessary JavaScript library. Since OpenDSA must insure that this information gets to all of the necessary parts of the system, there are certain places where support has been embedded. This section attempts to document them.

First, a given HTML page will need to load the MathJax library. Like all JavaScript libraries used by the system, these are enumerated in `tools/config_templates.py`, within the `html_context` variable. This will get it loaded into a module. Standalone AV and exercise developers are responsible for explicitly including it in their own HTML files if they want MathJax processing.

Next, MathJax will need some local configuration. This is added to module pages from the module page template at `RST/_themes/haiku/basic/layout.html`. Look for where it defines `MathJax.Hub.Config`. For standalone AVs and exercises, this is defined in `lib/odsaAV.js`. [TODO: Add in information about how KA infrastructure loads its own version of MathJax.]

Finally, in order to get MathJax translation to take effect within JSAV-controlled elements, JSAV has to be told to fire the translation on various events (`jsav-message` and `jsav-updatecounter`). This has been defined in both `odsaAV.js` and `odsaMOD.js`.

CHAPTER 18

QBank - Users Manual

Table of Contents

18.1 Introduction

QBank is a web application that assists Problem Authoring and Publishing. At the present time, this is an experimental system, and is not actively being used to create OpenDSA exercise content. But its goal is to replace a lot of the current programming required to develop our OpenDSA exercises. The user interface is meant to be intuitive and easy to understand. This document will give you a feel of the overall capabilities and functionality of QBank tool.

The **key** features of the QBank tool are:

- Easy interfaces for Problem authoring based on the main Problem types:
  - Static question – Multiple Choice Question
  - Dynamic question – Parameterised Question
  - Summative question – Multi-Part Question
  - Tool Specific question – Khan Academy Exercise Question
- Problem publishing - Parsing options to convert authored questions to different formats
  - Comma Separated Format - `csv`
  - Khan Academy Exercise Format
- Standard Authoring Interfaces based on the **Formal Problem Definition** for different Problem Types.

18.2 Key Terms

18.2.1 Problem Definition

The following lists out the essential components of a Problem.

Problem Statement Includes a function that generates a *Problem Instance*.

User Interface A mechanism that a user interacts with a create a *Student Answer*.

Model Answer Generator A function that takes a *Problem Instance* and generates a *Model Answer*.

Answer Evaluator A function that compares the *Student Answer* to the *Model Answer* to determine whether the *Student Answer* is correct or not.

Variables

These carry information from the *Problem Statement* to the *Model Answer Generator*.

18.2.2 Problem Types

1. Multiple Choice Question
2. Parameterized Question

- Variable that take a *List* of values
- Variable with values that vary over a *Range*

3. Multi - Part Question
4. Tool Specific Problem Authoring – Khan Academy Exercise Format

18.3 Write a Problem

18.3.1 Overview

The QBank editing interface consists of text boxes and buttons that are self explanatory. The text boxes also accepts HTML and JavaScript when appropriate.

The `What's this?` button gives helpful indicators on the purpose of different text boxes and what different parameters can be added to make the Problem powerful. It also tells you some functions that can be used to make more effective questions.

18.3.2 Problem

File Name

A unique identifier that is used to :

1. Store a problem in the database.
2. Refer to a problem in different parsed format
3. Refer to while creating a summative problem written from previously authored problems.

Difficulty

It just classifies problems as easy , medium or hard. This information can be used by a Smart tutor which bases the next question posed to the user on the correctness of the previous question.

If correct, a question with a higher difficulty is posed or vice-versa.

18.3.3 Variables

This allows for generation of different *problem instances* based on a static *Problem Template* with *variables* that take on different specified values.

Variables are used by specifying the *variable name* within `<var>...</var>` delimiters.

Variable Name is an ID for the var as that'll be the name that you'll refer to in the future.

Variable value Values that the variable can take is specified here. This can be as simple as commma separated values or functions that can be accepted by the publishing tool/ parsed into a compatible format.

For example:

```
<!-- Numbers from 1-5 -->
Variable Name : A
Variable Value : "1", "2" ,"3" ,"4", "5"
```

Another example to make a variable named SPEED1 that is a number from 11 to 20 you would do:

```
Variable Name : SPEED1
Variable Value : randRange(11,20)
```

The content of a `<var>...</var>` block is executed as JavaScript, with access to to all the properties and methods provided by the JavaScript Math object, as well as those defined in the modules/scripts you included:

```
<!-- Random number -10 to -1, 1 to 10.-->
Variable Name : A
Variable Value :(random() > 0.5 ? -1 : 1)*(rand(9) + 1)
```

Most mathematical problems that you generate will have some bit of randomness to them (in order to make an interesting, not-identical, problem).

Variable Reference

Use a `<var>...</var>` delimiters to refer to predefined variables while defining other variables or within other components of the problem.

For example in the following we define two variables (`AVG` and `TIME`) and then multiply them together and store them in a third variable (`DIST`).

```
<!-- Defining a variable using predefined variables. -->
Variable Name : AVG
Variable Value : 31 + rand(9)
Variable Name : TIME
Variable Value : 1 + rand(9)
Variable Name : DIST
Variable Value : AVG * TIME
```

18.3.4 Solution

The solution consists of the answer.

Answer

The answer can be any of the following.

1. A valid choice
2. A function that is the calculation of a question (with specified values for variables)
3. A previously defined variable.

   For example:: Answer : <var>round(DIST1)</var>

18.3.5 Choices

This can include text which acts as distractors for the user. The choices can also use the previously defined variables.

18.3.6 Hints

These are textual suggestions to help the user figure out the correct answer. The hints can also use the previously authored variable .

The hints are optional.

18.3.7 Scripts

The author can add scripts written in javaScript to add different functionality to the question.

This can add extra interactivity to the exercise.

This is optional as well.

18.3.8 Common Introduction

The problem overview/introduction is defined for a *Summative Problem*.

This is useful since the problems combined together can have some information that isn't explicitly part of the statement of the
For example, a Physics problem may describe the situation and the various objects in the world before asking about a certain quality of a certain object.

18.3.9 Problem Name

This part of the Problem Template defined for *Summative Problems*. The **Problem Name** is the file name of previously authored questions , that can be grouped together.

The `"question"` in a summative problem is just a file name.

18.3.10 Available functions and Tips

Generating Random Numbers

You can use random(), or one of the following methods defined in the math.js module (which should be included in all exercises):

1. randRange(min, max) - Get a random integer in [min, max].
2. randRange(min, max, count) - Get a random integer between min and max, inclusive. If count is specified, will return an array of random integers in the range.
3. randRangeUnique(min, max, count) - Get an array of unique random numbers between min and max, inclusive.
4. randRangeExclude(min, max, excludes) - Get a random integer between min and max, inclusive, that is never any of the values in the excludes array.
5. randRangeNonZero(min, max) - Get a random integer between min and max that is never zero.
6. randFromArray(arr) - Get a random member of arr.

18.4 Problem Type Specifics

18.4.1 Parameterized Question – List

Used to write a question with variables that take values that need to be specified explicitly

Variables: The values are specified within **double-quotes** and are **comma-separated**.

Show/Hide Variable Combination: It shows the various combination of the different values of variable can take.

This indicates the position of the answer for the particular combination has to be stored in the answer array.

Solution: The answer takes a one dimensional array, where the index corresponds the answer specified to a row on the table that shows the Variable Combination.

18.4.2 Parameterized Question – Range

This type of question is for math problems where there are calculations involved as the solution.

Variables: The values the variables take are between a range of values that can be specified using the javaScript `randRange(min,max)` function.

Solution: The answer is the exact calculation that is specified in the Problem Template.

For example:

```
A + B - C, is enough to specify the answer.
```

The author doesn't need to explicitly make the calculation and write the correct answer.

This ensures the validity of answer and gets rid of the risk specifying an incorrect answer for the solution.

Another important feature of this type of problem is, the author doesn't need to explicitly provide choices, since the user is expected to *fill the answer in the blank* provided. This is very effective for math problems.

18.4.3 Multi-Part Question

It is comprised of different previously authored questions from the Repository. You can combine different type of questions, free-form and multiple-choice simple questions and matching questions in your multipart question.

Common Introduction The problems share a common introduction which generally contains information that is common to the questions.

Problem

Problem Name Specify the exact identifier for a Problem. You can Browse the Problems in the repository by clicking the show button. You can then click *Add* and the Problem Name gets added.

18.4.4 Tool Specific Question – Khan Academy Exercise

This type of problem supports inputs that can be handled by Khan Academy.

Check out Khan Academy Exercise Framework Documentation.

Also, the `What's this` button gives a lot of direction in assisting an author while authoring a problem.

18.5 Publishing a Problem

The current version of the QBank supports **two** main publishing formats.

18.5.1 CSV format

Every authored exercise can be exported as a comma-separated file.

18.5.2 Khan Academy Exercise Format

The exercises can also be exported in a format fully compatible with Khan Academy.

18.6 Search for a Problem

This allows the author to browse previously written problems.

The author can:

1. Edit the problem.
2. Download the problem in CSV
3. Parse the problem in to Khan Academy Exercise Format.
4. View the problem in Khan Academy.

18.7 Helpful Hints

CHAPTER 19

QBank - Developer's Manual

Table of Contents

19.1 Overview

19.1.1 Introduction

QBank is a web application that assists Problem Authoring and Publishing. The user interface is intuitive and easy to understand. This document will give you a feel of the overall functionality of QBank tool and how to contribute to extending the code base.

19.1.2 QBank Features

Main Features Of QBank

Easy interfaces for Problem authoring based on the main Problem types:

- Static question – Multiple Choice Question
- Dynamic question – Parameterised Question
- Summative question – Multi-Part Question

- Tool Specific question – Khan Academy Exercise Question

Problem publishing - Parsing options to convert authored questions to different formats

- Comma Separated Format - `csv`
- Khan Academy Exercise Format

QBank is extremely flexible and extensible

- All display items are driven by templates using the powerful Django templating language
- All urls can be custom configured to your desired naming convention
- The formats that the questions can be parsed into can be extended to support specific needs of various publishing tool.
- Front ends specific to a Tool can also be easily extended from the existing(Khan Academy) interface.

Standardised Authoring Interfaces based on the **Formal Problem Definition** for different Problem Types.

19.2 Requirements

QBank is developed on the Django framework, therefore you do need a fully functioning Django instance The Django installation guide will step you through the process.

Please use Django 1.5.x at this time.

QBank requires Python 2.6.5 or later and a database supported by Django.

19.2.1 Directory Structure

The basic QBank directory structure looks as follows

```
.        `-- QBank
            |-- manage.py
            |-- settings.py
            |-- urls.py
            `-- templates
            `-- qtool
               |-- admin.py
               |-- models.py
               |-- forms.py
               |-- urls.py
               |-- views.py
               |-- tests.py
               `media
                   `Exercises
```

1. The `templates` directory contains all the HTML files for the tool.
2. The `media` directory contains the JavaScript, CSS, and other important files that are served by Django which are used by the QBank tool.
3. The `Exercises` directory in the `media` folder holds the generated exercises in the Khan Academy Exercise format that can be used as-is fully compatible to Khan Academy.

Run `qtool` app

Start it with ”./manage.py runserver” and browse to http://localhost:8000/qtool/

19.3 Installation

19.3.1 Setup

1. Check out the latest version of QBank into /home/user/src/:

   ```
   git clone
   https://github.com/annp89/QBank.git
   ```

   Note: This copies all the files in the QBank directory into `src` folder.

2. Install QBank onto your system:

   ```
   cd /home/user/src/qtool/
   sudo python setup.py install
   ```

   Note: An alternative to running the install is ensuring that /path/to/QBank/qtool is on your python path. This can be done by modifying the `PATH` variable to point to the correct file location.

3. Once the above step in completed, a local database can be created for your copy:

   ```
   cd /home/user/src/qtool/
   sudo python manage.py syncdb
   ```

19.3.2 Configuration

After having cloned the QBank repository, you have to make some changes in the `settings.py` file to work on your local machine.

- Ensure that `DJANGO_SETTINGS_MODULE` environment variable is set to `<project_name>.settings`; e.g. `QBank.settings`

settings.py

`TEMPLATE_DIRS` should point to a relative location where the templates folder is located.

```
TEMPLATE_DIRS =
("/<project_name>/templates",)
```

urls.py

The `media` folder contains all the javascript, css and other image files that are accessed by the tool.

```
url(r'^(?P<path>.*)$',
   'django.views.static.serve',
   {'document_root':
   '/quiz/qtool/media/'})
```

For viewing in Khan Academy Exercise Format, there are many `javascript` and `css` dependent files that have to be located in the media folder.

Django serves the static files in the above location since the `url` is specified with that particular regular expression.

19.3.3 View the Authoring tool

After you have completed the installation and the configuration, you can view the QBank tool using the commands below.

1. Start up the sample webserver to see your store:

   ```
   python manage.py runserver
   ```

2. In order to see your sample store, point your browser to:

   ```
   http://127.0.0.1:8000/qtool
   ```

3. If you want to see the admin interface, point your browser to:

   ```
   http://127.0.0.1:8000/admin
   ```

4. Additional detailed documentation can be found here:

   ```
   http://127.0.0.1:8000/admin/doc
   ```

Note: The above urls will be dependent on your Django setup. If you're running the webserver on the same machine you're developing on, the above urls should work. If not, use the appropriate url.

19.4 Developing a Tool-Specific Authoring Interface

Files that need to be manipulated in order to develop a tool specific to the interface

1. `forms.py`:

   A new form specific to the interface be written by extending the basic `'Problem Form` by either adding/ excluding fields to it.

   ```
   ``SimpleProblemForm`` was created to handle simple Multi Choice Questions.
   ```

2. `views.py`:

   For each type of form, a view (class definition) needs to exist to render the template specific to that particular form.

   ```
   e.g. ``simple`` for is a view for ``SimpleProblemForm``
   ```

Another view needs to exist for parsing the form into a desired publishing format.

```
e.g.
1. ``write_csv`` to parse the authored Problem to a ``csv`` file.
2. ``ka_gen`` to parse the authored Problem to a ``Khan Academy Exercise format``
```

19.5 Defining another Publishing format

In the `views.py`, the parsing of the content entered in the Frontend Interface into another format is done in the `ka_gen` function. This function essentially takes all the content fed into the Front end and parses it with the additional information that converts it into a particular format.

- For instance, to write exercises in the *Khan Academy exercise format*, the content has to be converted into an `html` page. All the appropriate header files, scripts, `css`` and images have to be correctly linked to the question based on the options selected and the type of question being authored.

The *ka_gen* class parses the content and writes a file which is stored in the `media` folder. This allows the author to simply take the parsed file and plug it into the desired tool.

19.6 About the project

QBank - Problem Authoring Tool, was developed by **Ann Paul**, Masters student, Department of Computer Science, Virginia Polytechnic and State University.

This project was done under the supervision and able guidance of **Dr. Cliff Shaffer**, Professor, Department of Computer Science, Virginia Tech.