Scalable Agenda Services

Alex Vandiver
alexmv@mit.edu

May 20, 2005

Abstract

A web-based application was written to provide a scalable and user-interface oriented tool for generating agendas for visitors to MIT. To do so, it uses common tools for rapid development, including templating, database abstraction, and common open-source software.

1 Introduction

In an academic setting, it is not uncommon for professors to visit other universities and speak to their colleagues. However, the logistical overhead of determining the times when professors are free to meet, and coordinating the various times and locations involved, can be a significant hassle.

Currently, administrative assistants email out to interested professors, asking for times when they are available. They wait for enough responses, then attempt to merge the information together and establish a schedule that works for all involved. This can be very time-consuming.

The purpose of this project was to translate this process into a centralized agenda server, which would be responsible for aiding the organizer in laying out the framework, cataloging the responses, and presenting the information to the visitor.

2 Design

The planned use of the agenda tool is best described with a use case. The following sections describe a hypothetical use case between Alice the Administrative Assistant, Victor the Visitor, and Peter and Pam the Professors[1].

2.1 Planning

In this stage, Alice has been notified through email and word-of-mouth that Victor will be visiting the campus to give a talk on the cryptographic security of Widgets. She emails Victor, asking for the abstract of his paper, or (if possible) the paper itself. In the meantime, she points her browser at `https://agenda.csail.mit.edu/`.

The browser automatically recognizes her as Alice based on her client-side certificate. It presents her with a list of agendas she has made recently, and the option to create a new agenda.

Alice selects the latter, and is presented with a screen to input who is visiting. As she types Victor's name, the system proposes guesses based on previous visitors. As Victor has not visited MIT before, she is automatically prompted to fill in basic details about him, including first name, last name, and email address. Victor receives an email telling him that an account has been created for him by Alice on the agenda system; it

[1] Though this paper refers to them throughout as "professors," the agenda software holds just as well for *any* kind of host at, or visitor to, the Institute.

includes a randomly generated password.

Alice chooses the appropriate date for the visit from a calendar pop-up. The screen then advances to a blank schedule for the day. Alice chooses to schedule an event named "lunch," enters 12-1 as the time, and begins to type "stata dining" into the location field. As she begins typing, a dialog pops up beneath the input box and suggests "Faculty Dining (32-G401)," which she selects using the arrow key. Alice enters a quick description, then adds the event to the schedule. She also adds a "presentation" from 10-11pm, and sets a location.

Having blocked out the immovable parts of the schedule, Alice adds several 45-minute "slot" events in the morning and afternoon. She then searches for and adds the professors who she believes might be interested in meeting with Victor to the list of parties interested in the slots. Having done all of the scheduling which she can at this time, she clicks "release schedule."

2.2 Responding

The action of releasing the schedule automatically sent emails to all of the faculty that Alice listed, informing them that they should select report their availability if they are interested in meeting with Victor. Both Peter and Pam get these emails.

Pam checks her mail first and clicks on the link in the email. She is presented with a page detailing all of the slots that Alice set up, and asking her to show which of the slots she would be able to meet with Victor – the system already knows who Pam is, based on her client-side certificate.

Pam notes that she is available during the 1:00 - 1:45 and 1:45 - 2:30 slots; as she does this, the system prompts her for a location for each, with a default of her office. She accepts these defaults and saves her preferences.

Meanwhile, Alice has gotten email back from Victor, who has attached the paper he will be speaking about. Alice visits the agenda and uploads the paper, which is then viewable by everyone.

Peter remembers the agenda when he doesn't have access to his email; luckily, his browser has a certificate. He points it at `https://agenda.csail.mit.edu/`, which recognizes him and tells him he has one agenda which is waiting on his availability. Peter reads the paper by following the link at the top of the agenda, and decides he does want to meet with Victor. He checks off 11:15 - 12:00 and 1:00 - 1:45, but specifies the conference room 32-346 instead of his office, which is a mess.

2.3 Scheduling

The next day, Alice checks back and notes that she has a number of responses to the agenda, and decides to begin scheduling. For each time slot, she selects one of the professors who announced availability for that period; thus each slot is filled.

Happy with the schedule, Alice clicks "finalize schedule." This sends the completed schedule and a URL out to all of the participants (Alice, Victor, Peter, Pam, and any other professors) which they may click to see the schedule as it stands. This URL will continue to work in perpetuity.

2.4 Viewing

Victor gets the email, and follows the link. He is prompted for his username and password; he enters these based on the information he received in the earlier email. He is then presented with a view of the completed schedule, including maps of each location.

3 Implementation

The agenda application is a database-backed website written in Perl using common Perl templating and database abstraction modules, based on information extracted from MIT's Data Warehouse. Each technology used in the system is presented below, followed by a detailed breakdown of the individual components and their function.

3.1 Hardware and kernel

The application is hosted on a Celeron 2.6Ghz with 512M of RAM, and a 38G hard drive. It is running Gentoo Linux, a distribution with a rather robust and up-to-date package system. The package system, known as `portage`, builds all packages from source. This results in a system with less dependency problems than many other distributions, as executables are compiled against libraries that are actually present.

The primary maintenance and installation tool used by `portage` is called `emerge`. At its simplest, `emerge` *`packagename`* installs the named package onto the system.

3.2 Apache

The entire application runs under the common Unix webserver Apache. Apache was chosen because it is a portable, scalable web server that is available for free. Additionally, Apache has a large number of modules that can be added to the webserver to customize the server for particular applications.

Apache 2.0 is suggested as the "best available version" by the Apache Foundation; however, the author has experienced numerous small bugs with the 2.0 series, mostly due to incomplete testing of modules such as mod_perl and OpenSSL (below). Because of this, the older and more stable Apache 1.3 branch was chosen.

3.2.1 mod_perl

Usually under Apache, the process of generating dynamic content is done via a module known as mod_cgi. This is the well-known Common Gateway Interface which was originally invented by NCSA for the NCSA HTTPd web server in 1993, and has changed very little since then. Under mod_cgi, a request for a dynamic web page causes the webserver to fork another process that runs a particular file, sending it any parameters via environment variables, and sending its output to the waiting browser.

This process is inefficient for several reasons: the process which generates the page cannot maintain any state between requests. If it needs to load a large amount of data before replying, it will pay that time cost upon every request. Additionally, it takes time and memory to fork the process and start the process itself. When more and more users demand faster and faster response times from a single host, every microsecond counts.

mod_perl is a solution that attempts to remove several of the time costs of standard CGI, by embedding a persistent Perl interpreter inside Apache. By doing this, the overhead of starting a Perl process occurs just once (at server startup) instead of upon every request. Additionally, persistent state (such as database connections) can be maintained between requests, further increasing the rate at which pages can be served.

The persistent interpreter also allows the programmer to approach the application from a more coherent view – as opposed to a large number of stand-alone programs, the entire request phase is handled by *one* control path. This helps make the code more centralized and maintainable. Though it was not used in this project, mod_perl also allows Perl code to be run at any phase of the Apache request cycle, including URI translation, access control, authentication, authorization, content generation, and logging.

For the purposes of this project, mod_perl was

compiled as a Dynamic Shared Object (DSO), a kind of loadable module. In this case, the installation was a simple matter of using the Gentoo `emerge` tool, described above, by typing `emerge mod_perl` at a root prompt.

3.2.2 OpenSSL

A key concept in the authentication scheme that the agenda server uses is that of an SSL personal certificate. An SSL personal certificate is a unique identifier that states a certain certificate authority's view of who you are. That is, if one trusts the certificate authority, the personal certificate provides a unique designator of who the presenter of the certificate is.

Because a large portion of the audience of this project is affiliated with MIT in some way, they all possess SSL personal certificates, signed by MIT's certificate authority. By obtaining the certificate authority's public key, the webserver can verify the validity of all personal certificates signed by MIT's certificate authority. This allows the webserver a simple means to authenticate a large cross-section of the users of the system.

Happily, there exists a common implementation of SSL, known as OpenSSL, which has a module (creatively named mod_ssl) which allows Apache to communicate over SSL. Much like the installation of mod_perl above, installing the mod_ssl DSO was simply a matter of running `emerge mod_ssl`

It is worth noting that the certificate authority which signs the server's certificate, and the certificate authority that signs the users' certificates, are two different entities. In the case of `agenda.csail.mit.edu`, the server's certificate is signed by `ca.csail.mit.edu`, which identifies itself as "MIT Laboratory for Computer Science, SSL Servers, SSL Server CA." The client certificates, on the other hand, are signed by `ca.mit.edu`, which identifies itself as "Massachusetts Institute of Technology, Client CA v1."

3.3 MySQL

The simplest way to track large quantities of data is in a relational database. There are a large variety of relational databases available for free for the Linux platform, the most common being PostgreSQL, MySQL, and Oracle. Oracle is a much more feature-rich database server, however, as in many cases, it was simply overkill for the size of the database that would be needed. The *minimum* requirements for an Oracle 10g server are 512M of RAM, a 1Ghz processor, and 6.5G of free disk space. As those are very close to the specifications of the hardware in question, Oracle was not a viable option.

The argument between PostgreSQL and MySQL is a hotly contested one; the speed and feature set of PostgreSQL is generally admitted to be slightly larger, but MySQL has a larger market share. Purely for reasons of familiarity, MySQL was chosen as the database server for this project. However, due to the database independence layer (described in 3.5, below), this is a easily changeable decision.

The MySQL engine was installed using `emerge mysql`. A number of tables were created in the `agenda` database; their columns and relationships are shown in Figure 1.

3.4 Data warehouse

In order to provide the application with as complete a repertoire of people and locations, a large section of data was loaded from MIT's Data Warehouse. The Data Warehouse is a large Oracle database that contains information about students, employees, finances, building and room information, and many other large data sets. This data, while large, and running under Oracle, suffers from slow query times, which prohibited direct run-time look-ups to the Data Warehouse.

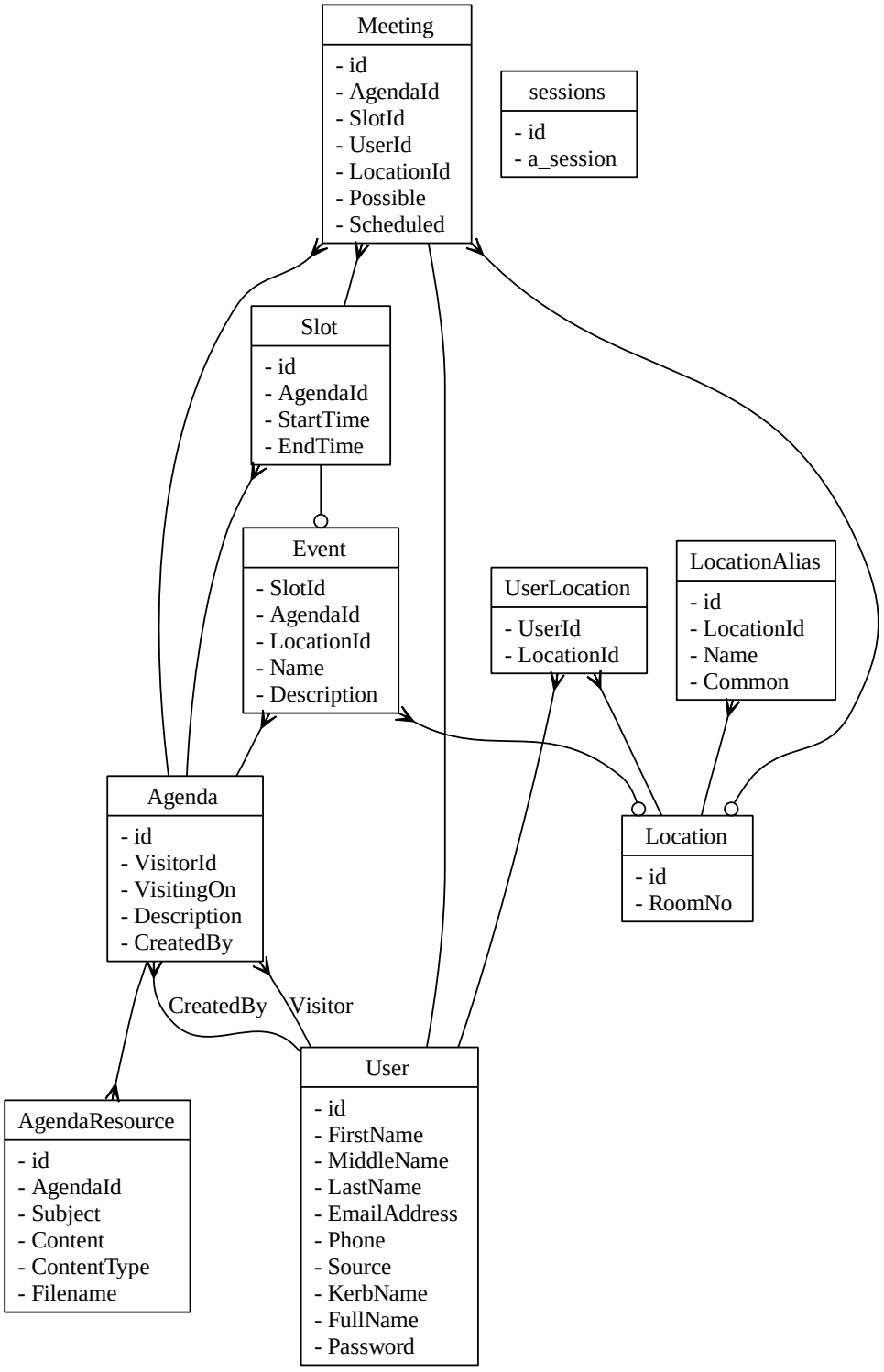

Figure 1: The database's tables, their columns, and their inter-relationships. Relationships are implemeted using standard ER symbols; a circle denotes an optional relationship, a crow's foot denotes a "many" relationship.

Thus, the student and employee information from the Data Warehouse was imported into the local MySQL server, and inserted into the User table. Additionally, the information about the offices of the employees was inserted into the Location table, and the UserLocation table was used to record the relationship between office and office occupant.

These local tables could also have more pertinent indexes added to them, reducing nearly all database look-ups to being index look-ups rather than sequential scans. Sequential scans are database queries which require the database to analyze each row of the table one at a time, yielding a O(n) running time; index scans, by comparison, are O(1). Obviously, whenever scaling is possibly an issue, index scans are preferable to sequential scans.

3.5 Class::DBI

When writing application-layer code, inserting the underlying syntax of the SQL required to retrieve data from the database is a large hassle, requiring mixing large chunks of SQL within large chunks of Perl. This forces the reader of the code to understand what the SQL is doing to understand the Perl, as well as forcing a large degree of repetition among the SQL code. Additionally, if the SQL is specific to the database sever that it runs on, changing database servers becomes a long an arduous task.

The solution to these problems is a *database independence layer*, or a *database abstraction layer*. This serves to separate the code that deals with and understands the database, from the code that wants only to deal with the data in an abstract way. The most common way to do this is to treat rows in the database as objects, whose accessors and mutators fetch and update columns of the table.

One module to accomplish this in Perl is `Class::DBI`. `Class::DBI` creates an object class for every table in the database, and automatically constructs appropriate methods on the class to allow for reading and updating the columns of the class – without having to ever write any SQL. Additionally, if Class::DBI is told of the relationships between the tables, it can construct methods that *join* the various tables that relate, allowing accessor methods that return other objects in other tables. In this way, Class::DBI allows us to abstract nearly all SQL out of the main methods of the program, and instead program using a sophisticated object-oriented API.

3.6 `startup.pl`

The file which loads `Class::DBI` and informs it of the relationships between the tables is called `startup.pl` It is so named because it is loaded by the Apache server upon startup. By loading the modules and the database configuration at startup, parts of the memory used can be shared across the multiple Apache processes, reducing the overall memory footprint.

3.7 Mason

The bulk of the logic of the system lies in a collection of files under the document root of the webserver. These files are not simply html files; they are part of a templating system known as Mason. Mason, like all templating languages, works to allow for code reuse and comprehension by splitting off reusable sections, and by separating the generated content from the computation.

The latter is part of a software architecture approach known as Model, View, Controller (MVC). MVC applications attempt to split the data representation (the model), the presentation of information (the view) and the aspects that can be manipulated (the controller). The agenda application attempts to hold to MVC principles; the `Class::DBI` database abstraction layer serves as the model, and the Mason templates serve to separate the HTML view from the

Perl controller.

The following sections attempt to give a brief overview of how Mason works. A much more thorough discussion may be found in *Embedding Perl in HTML with Mason*, By Dave Rolsky and Ken Williams[2].

3.7.1 Components

Mason calls each file on disk a "component." At the most basic level, a component is called every time a request is made. *Which* component is run is resolved relative to the document root, identically to how a file would be served, were Apache running without mod_perl and Mason.

The control flow inside the component is more complex, however. The first section of the component which is run is the the <%init> block, so called because it is comprised of all of the Perl code between <%init> and </%init> tags. This code generally fetches objects from the database, possibly modifies them, and organizes them.

It often depends on the parameters that were passed in via the GET or POST request to the webserver; the <%args> block exists to simplify accessing parameters. If a parameter exists that matches a line in the <%args> block, then that variable is set to the parameter's value. That is, if the url was `http://agenda/foo?bar=baz` and the <%args> block contained the line `$bar`, then the variable `$bar` would be set to the string "baz." This simplifies argument access.

After the <%init> block has been run, the main body of the component is processed. By and large, this consists of HTML, which is output directly as it appears. There are three common exceptions: <% *expression* %> tags, <& *component* &> tags, and line beginning with a %.

The <% *expression* %> tag is used to insert the value of a Perl *expression* into the page as it is generated. The <& *component* &> syntax is used to call another component, and insert its output at that location. This is commonly used to factor out reused sections of code into separate files. It is also possible to pass arguments to the component, by using the form <& *component, argument => value* &>. Lastly, lines which start with % are processed as lines of Perl. This is most commonly use to insert looping and control structures.

3.7.2 autohandler

There are two special components that Mason uses. One such is the "autohandler." This component is run at the very beginning of every request, before the actual page itself is run. This allows the autohandler to do any global setup that is required, to check authentication, and so on. In the case of the agenda application, the autohandler is primarily used to check for authentication and to set up the session.

3.7.3 dhandler

The "dhandler" is the other special component; in this case, the "d" stands for "default." The dhandler is run when Mason cannot find the component that it is looking for. If a component is missing, Mason looks in the directory to find a dhandler – if found, the dhandler is run instead. This allows Mason to intercept what look, to the browser, like normal requests, and generate files and responses on the fly. The agenda application uses this to transparently pull attachments out of the database.

3.7.4 Caching

Whenever large databases are involved, there is always a danger of perceptible delays due to complex SQL queries. In many cases, however, the

[2]It should be noted that the full contents of this book are available from any computer on MIT's network, via http://safari.oreilly.com/JVXSL.asp?xmlid=0-596-00225-4

responses to the queries are relatively static, and thus benefit immensely from caching. Mason possesses a powerful set of tools to allow the output of components to be cached, with optional durations, keys, and expiration handlers.

In the agenda application, caching was used aggressively whenever possible, to reduce server load times. The most obvious such use is in the "live search" results – the list of users in the system is relatively static, as is the set of locations that the system is aware of. Because the "live search" results in a large number of queries to the server, often repeating the same query, it is a prime candidate for caching. The other key use of caching is when generating images of rooms from the Facilities map server, which requires querying the server for the location of the room before displaying it. As rooms' locations will not change over time, this is another obvious candidate for caching.

3.8 AJAX

Another technology which was used throughout the application is what is coming to be termed "AJAX," which stands for "Asynchronous JavaScript and XML." AJAX is a combination of HTML and CSS for presentation, JavaScript and the Document Object Model (DOM) for interactivity, and a function known as XMLHttpRequest to allow the browser to fresh *part* of a web page at once.

AJAX-type applications include Google's Gmail, Google Maps, and Google Groups – though Google calls the technology that drives these applications "JavaScript" rather than "AJAX." The agenda application uses AJAX to allow for listing of real-time results to searches, allowing easy completion of half-typed usernames and locations.

3.9 Workflow and file description

The following is an in-depth description of the function of every component of the agenda application. An attempt has been made to group components which refer to each other in close proximity.

There are two overall naming trends; first, any file which is not meant to be accessed directly by the browser is placed in a subdirectory named "Elements." Additionally, files are placed in a directory named after the *kind* of object that they apply to.

3.9.1 /autohandler

The autohandler is called for every page request. This allows it to do page processing like checking for authentication, among other things.

First, the browser cookies are retrieved, and used to set up a persistent session in the database. Next, authentication is checked via one of two methods: certificate or password. If a certificate was provided, mod_ssl has set a number of environment variables, which are read and parsed to load a user. If a valid certificate was provided, but no accompanying user is found in the database, the user is redirected to a /Elements/CreateLogin whereby they can enter their pertinent information.

For password authentication, the user is presented with the contents of /Elements/Login, and asked to enter their username and password. When the username and password are submitted, the autohandler loads the appropriate user to verify the password and finish authentication. Note that users that were imported into the system from the Data Warehouse (see section 3.4) *cannot* log in until they set a password; users that were created by other users start with a random password.

Note that all of this is done *for every page*, allowing authentication to happen at any point if it is needed. Additionally, the URL of the request is

preserved, allowing the user to log directly into an arbitrary part of the site, saving them clicks.

3.9.2 /Elements/Login

This component is called by the autohandler when authentication is needed. Note that the action for this form is whatever URL was originally submitted, and we pass through all of the original form parameters when we submit – allowing this login process to be transparent.

3.9.3 /Elements/CreateLogin

This is called when the user has a certificate, but doesn't exist in the database; they are a valid user, but not one we know of yet, so we ask them for some information. If we see that the information we need was just submitted, then we create the user and return it.

3.9.4 /Logout.html

Logs the user out, by clearing the session, then redirecting to the http://agenda.csail.mit.edu/. The redirect is necessary to keep the user from being logged right back in, if they logged in using certificates.

3.9.5 /Elements/Wrapper

This outputs the standard HTML header and footer, including linking to the CSS file, the JavaScript files, as well as providing a title and heading.

3.9.6 /Static/styles-site.css

The CSS file containing all of the styles used in the site. CSS is used to alter the fine details of the presentation of the site; it also sets the color scheme, amongst other things.

3.9.7 /index.html

The front page, that most users see as soon as they arrive. Almost all of the work is done by one of several sub-components; listing slots the user has yet to respond to, showing agendas the user created, and agendas that were created for the user. These correspond to the three major user groups of the application: professors, administrators, and visitors.

3.9.8 /Elements/MyAgendas

This component lists all of the agendas that the current user has created. Thus, this component is mostly for administrators.

3.9.9 /Elements/MeetingTimes

Shows all of the time slots that the user has yet to confirm their availability for, and allows them to set their availability and desired meeting location; this component is primarily seen by faculty. Most of the work of showing the forms to *update* the information is done by /Agenda/Elements/Availability, below.

3.9.10 /Agenda/Elements/Availability

This component looks at the arguments that were passed to the top-level component, and updates the availability of the user for the given meeting. It also displays the form needed to update the meeting.

3.9.11 /Elements/Invitations

Creates a list of agendas that are for the current user; that is, agendas where the current user is the visitor. Additionally, the organizer in chage of each such agenda is shown.

3.9.12 /Users/View.html

Shows public information about a particular user. If the user in question is the current user, they are redirected to their preferences page.

3.9.13 /Users/Preferences.html

Allows a user to edit their personal information and change their password, as well as location information about where their offices are.

3.9.14 /Elements/Next

Implements a poor man's continuation scheme. That is, the user's session is used to record what webpage the user should see next. When there is an interrupt, caused, for example, by a user or location not existing when the administrator expects them to, the expected next location is pushed onto the session's stack using /Elements/AddNext, and the browser is redirected to create the user or location. When the user or location has been created, they call /Elements/Next, which pops the location off of the session's stack, and redirects the browser to it.

3.9.15 /Elements/AddNext

The counterpart to /Elements/Next; takes a location to return to, and the name of the parameter to pass the return value in, and pushes it onto the session's stack.

3.9.16 /Users/New.html

Prompts for details on a new user, generates a random password for them, and sends them email. This operation is meant as an interruption of an in-progress operation, such as creating an agenda. As such, after creating the user, it redirects to the original destination – see /Elements/Next, below.

The email that is sent to the new user is in /Users/Elements/NewUserEmail

3.9.17 /Users/Elements/Password

Generates a new, random password for a user. This is called when a new user is created.

3.9.18 /Users/Elements/NewUserEmail

The body of the email that is sent to new users. It includes a brief introduction to the system, information on what user created the account for them, and their email address and password.

3.9.19 /Location/View.html

Allows the user to see a map to a given location, as well as what users list that location as one of their offices. Users may add or remove themselves from the location, using /Location/AddUser.html and /Location/DelUser.html as well.

3.9.20 /Location/AddUser.html

Adds the current user to the given location, and then redirects back to /Location/View.

3.9.21 /Location/DelUser.html

Removes the current user from the given location, and then redirects back to /Location/View.

3.9.22 /Location/Elements/Image

Returns HTML suitable to insert an (approx) 600x600 image. This is garnered from either the Building 32 CSAIL maps, or Facilities' map server ims.mit.edu. To speed up performance, the component caches the results indefinitely, on the assumption that rooms do not move over time.

3.9.23 /Location/New.html

Creates a new location with the given name. Whenever a location is created, it must somehow be associated with a room number. Two ways are provided to do this: entering the room number, or searching for another alias for the room, whose room number will be used instead.

3.9.24 /Elements/LiveSearch

Adds a dynamic search element to an existing form. As the user types into the text area, a real-time list of possible completions appears beneath the box, allowing the user to select one of them using the arrow keys or mouse. This is done using AJAX, but the text area that is inserted is fully functional even in browsers that do not support JavaScript.

The JavaScript uses XMLHttpRequest to send a request to a server element (usually /Elements/Lookup, which searches for users) that returns an HTML list of results, which is then inserted into the HTML page at the appropriate location. It is also possible to request that the search element search rooms and their aliases, instead of users.

3.9.25 /Elements/Lookup

This is the code which is called server-side during a LiveSearch. It searches through users, looking for those whose name or email possibly matches the given substring. The results are returned in an HTML list.

3.9.26 /Elements/Where

This elements is called internally by LiveSearch when the "rooms" option is specified. It does a SQL query to find possible locations that match the user's query, and returns an HTML list of possible matches.

3.9.27 /Static/livesearch.js

This JavaScript files implements the client-side aspects of LiveSearch. It auto-detects all of the LiveSearch entries on a page, and adds key bindings to all of them. These key bindings fire events which start XMLHttpRequest connections to the server to update the drop-down completion box in real time.

3.9.28 /Agenda/New.html

Prompts the user for the date and description of the visit. The date selection is done via the dynamic entry /Elements/PickDate, below. Though the date cannot be changed later, the description can be altered at any time by the agenda's creator.

3.9.29 /Elements/PickDate

Displays a form allowing the user to pick a date – a form which is AJAX-driven on browsers that support it, displaying a pop-up calendar for them to select from.

3.9.30 /Static/calendarDateInput.js

The client-side JavaScript that powers /Elements/PickDate.

3.9.31 /Agenda/View.html

This shows the schedule, including the description, any uploaded resources, and all of the agenda's time slots. The options available to the user from this screen depend on who they are. Visitors see only the meetings that have been scheduled by the organizer. Organizers of agendas see which professors have updated their availability, and are able to select which meeting will be scheduled. Professors see only their own availability, and are able to change it.

The actual display of the description, resources, and time slots is done via /Agenda/Elements/ShowDescription, /Agenda/Elements/ListResources, and /Agenda/Elements/ShowSlots, respectively.

3.9.32 /Agenda/Elements/ShowDescription

Shows the description of a given agenda, or "no description" if there is none. This is used by /Agenda/View.html as well as /Elements/MeetingTimes

3.9.33 /Agenda/Elements/ListResources

Gives links to the uploaded resources for a given agenda – links are to /Resources/number/filename. The files are served by /Resources/dhandler

3.9.34 /Agenda/Elements/ShowSlots

Show the slots of an agenda. If the user is the creator, then one can choose which meeting is scheduled. If one is the visitor, one sees what has been scheduled. If one has a meeting, they can select their availability and proposed location.

3.9.35 /Static/util.js

Helper JavaScript for /Agenda/Elements/ShowSlots and other components; this deals with actively hiding and showing parts of the page based on what radio boxes are chosen.

3.9.36 /Agenda/Elements/ShowMeeting

Displays information about a meeting, including who it is with, their availability, where it is, if it has been officially scheduled.

3.9.37 /Agenda/Edit.html

Allows the creator of the agenda to edit the description of and upload resources for an agenda. The resources are stored in the database, not in the file system.

3.9.38 /Agenda/Slots.html

Displays, and allows the user to edit, the list of possible slots that can have meetings put in them. The organizer can create slots for meetings, and set possible people who can meet during that slot, as well as scheduling events in specific locations. If an entered location doesn't exist in the database, a side excursion will be made to /Location/New.html to define it.

3.9.39 /Elements/ParseTime

A component which has no output to the browser, but which returns a date object to the component that called it. It is primarily used by /Agenda/Slots.html to parse the user-entered start and end times of slots.

3.9.40 /Agenda/Release.html

Sends email to all of the people who are mentioned in the agenda, announcing that they should fill in their availability. The body of the email is contained in /Agenda/Elements/ReleaseEmail.

3.9.41 /Agenda/Elements/ReleaseEmail

The body of the email that is sent to users who are mentioned in an agenda. It includes information on who will be visiting and when, as well as who is organizing the schedule.

3.9.42 /Agenda/Finalize.html

Sends email to all of the people who are scheduled in the agenda, as well as the visitor, announcing that the schedule has been finalized. The body of the email is contained in /Agenda/Elements/FinalizeEmail.

3.9.43 /Agenda/Elements/FinalizeEmail

The body of the email that is sent to users who are mentioned in an agenda. It includes the scheduled meetings, as well as a link to the agenda server for more information.

3.9.44 /Resources/dhandler

This component serves files that were uploaded by the organizer. The files are stored in the database; despite the real-looking URL that browsers see, the file never actually exists on disk. The dhandler is used to intercept attempts to access the file, and the correct resource is extracted from the table and given to the browser.

4 Conclusions

The application described above streamlines the agenda creation process, automating it and providing a central rendezvous point for all participants. It also serves as a testament to the rapid development abilities of the tools and technologies that were used.