MVSim Documentation

Release 2.0

CCNMTL

August 18, 2015

Contents

Contents:

CHAPTER 1

Overview

Aspects of MVSIM that should be considered in documentation:

1. The Logic Engine, a pure Python module which receives a Python object representing a game state and returns a new game state.
2. The Configuration system, which describes the interface to and from the logic engine, by providing schema for serializing and deserializing coefficient and variable values.
3. The User Input model, which, like the Configuration system, defines which variables may be edited directly by the user during gameplay.
4. The State model, which stores a single state of a game in a JSON formatted text blob, and which is deserialized according to a Configuration. The *created* timestamp of a state expresses the time when a user played a game turn that resulted in that state.
5. The Game model, which contains a set of States (implicitly ordered by their *created* timestamp) and a link to a single Configuration and a single User Input, which determine how the Game will interpret its States.
6. The Events system, which determines which notifications to display in the End Of Turn Report based on the game state before and after a turn was executed. Events are expressed as rows in a CSV file, and include information like:

   - Under what conditions should this event occur?
   - What text should be displayed to the user if this event occurs?
   - What is the event's severity?

   The Events are currently read directly from the "events.csv" file during runtime. It would probably make sense to move them to the database and associate them with new games in the same way that configurations are associated with games.
7. Starting States, which are just States associated with a Course Section. We need a UI for faculty to promote States to Starting States.
8. :doc'The Courseaffils integration and CourseSection model <course-sections>'.
9. The Graphing Tool.

Contents

CHAPTER 2

The Logic Engine

The logic engine is an effectively standalone, pure Python module that determines the new state of an MVSim game by receiving a set of Python objects that fully describe the game's current state, and returning back a new game state represented by a new set of Python variables and a boolean flag that signals whether the game is now finished.

The set of Python objects that constitute the interface into the logic engine is described by the configuration system. Currently, this consists of a list of variables and a list of coefficients; conventionally (and only loosely enforced in the code) the only difference is that variables will change value from turn to turn, and coefficients will not.

The logic engine's code lives in the `engine` subdirectory of the Django project. The main entry point and only real API exposed by the logic engine is `engine.logic.Turn.go()`; to see its usage look at the `main.views.submit_turn()` view which invokes it.

2.1 Events

There is actually an additional requirement besides coefficients and variables that must be provided to the logic engine: event notifications. These are currently provided to the logic engine "out of band" by reading a CSV file, but it's worth being aware of. For more see the events documentation.

2.2 Display Logic and Simple Controller

The functions in `engine.display_logic` are used by the views to build the necessary context dictionary to be passed to the game's various templates (game view, season report, game over report, etc)

The code in this module was gradually extracted from TurboGears Kid template files and moved into Python code. Feel free to refactor it further to make it reasonably clean and/or simplify the templates to not need so much of this stuff.

Similarly, the functions in `engine.simple_controller` were extracted from various places in the original TurboGears project. They are used in several places – engine logic code, Django view code and template tags.

2.3 Future Directions

2.3.1 Cleanup

First of all, there are a couple of pretty straightforward refactorings that would be worthwhile: the TwisterClient requirement should probably be replaced with local use of Python's stdlib random module, and the events should

probably live in the database and be passed to the logic engine rather than being read from a file in the source repository.

2.3.2 Pluggable Implementations

There may also be opportunities for broader refactoring of the code within the logic engine. I've already built pretty well encapsulated and self-documenting modules to represent fuel types (`engine.fuel`) and disease types (`engine.disease`) and I think other aspects of the simulation could similarly be encapsulated in logical units.

This sort of refactoring seems to make it possible to build pluggable systems within the simulation engine – writing a new fuel type or a new kind of disease is now pretty straightforward, although a few important details of UI presentation would need to be rearranged in order to make them completely pluggable. In conjunction with a system of multiple parallel configuration schemas (since the absence or presence of each type of disease or fuel impacts the set of required variables, coefficients, and events) this could be used to let faculty turn on and off individual diseases or fuel sources; or even to let faculty or students define their own diseases or fuel sources and plug them into the system. I'm really interested in pursuing this direction, and also in further refactoring the logic engine to find other domains that could undergo this treatment (village improvements and crop types come to mind)

2.3.3 Packaging

Related to the above point, I think it would make sense to actually remove the logic engine module from the Django project; move its required configuration data to live in that module; and ship the pair as an independent Python package with, effectively, no dependencies and no user interface. The Django project would then be (approximately) a simulation platform with pluggable logic implementations. Especially if users of the system were implementing custom instances of common types (fuel, food, disease etc) this decoupled packaging structure could make code-sharing easy and really interesting.

It would also theoretically allow for other consumers of the logic engine besides the Django platform, and *entirely* different logic engines within the same Django platform, but those outcomes both seem pretty far off and of no clear value.

Contents

CHAPTER 3

The Configuration System

The Configuration system allows administrators to build a schema by defining a set of typed Variables, and then specifying which of those Variables should be considered as "variables" (which change on each turn) and as "coefficients" (which are stable throughout a game) in the game. A Configuration serves two purposes:

1. It defines and documents the necessary interface with the logic engine. Executing a turn consists of submitting a set of variables and coefficients, whose names and allowed types are defined by a Configuration, to the logic engine; and receiving a new set of variables and coefficients back from the logic engine, representing the new state of the game.
2. It defines the schema by which a game state (represented as a JSON structure stored in a single text field) is validated and deserialized to a Python object that can be sent to the logic engine; and by which a Python object received from the logic engine can be serialized back into a JSON structure to be stored back in the database.

The Configuration system is implemented with Colander[1], a small standalone library by Chris McDonough that is maintained as part of the Pylons Project.

[1] https://docs.pylonsproject.org/projects/colander/dev/

3.1 Variables

The Configuration system has a notion of Variables (which are distinct from the logic engine's notion of "variables" vs "coefficients"; a Variable can be used by the logic engine as either a variable or a coefficient)

Variables are stored in the database (*main_variable* table) with a name and a type (e.g. integer, boolean, or list) – both as text columns. When constructing a Colander Schema out of a Configuration, the Variables' names and types are used to declare the elements of the schema.

Variables can also store "extra type information" as a JSON blob, which can be used to add additional information to the Colander Schema. At the moment, the only "extra type information" implemented are:

- "choices" (optional; should not be used for composite Variable types): provide a list of values to restrict the Variable's possible values to
- "listof" (required for List Variables; unused otherwise): provide a single string value that defines what kind of list this Variable is. Values can be either a Variable Type (int, list, string, etc) or the name of another Variable.
- "attributes" (required for Dict Variables; unused otherwise): provide a list of string values, each the name of another Variable, which this Variable contains.

Additional "extra type information" (e.g. ranges for numerical values) could be implemented by modifying the *mvsim.main.Variable.schema* method to interpret different parts of the field and modify the *colander* constructors accordingly.

3.2 Design Contrast

The system is an update of the TurboGears MVSim's system of Variables, Coefficients, Configurations, and SavedStates. The major differences are:

- A strict separation of schema and state. The previous incarnation had a much looser boundary between the two: Variables and Coefficients "schema" held default values which could be overridden by particular (Coefficient) Configurations and (Variable) SavedStates. Likewise, the previous incarnation merged its concepts of schema and "starting state" – the place where a set of variables and coefficients were given initial values for new games(Configuration and SavedState) was the same place where the set of required variables and coefficients for the logic engine's interface was implicitly declared.
- Merging Configuration and SavedState into a single "State" object that includes both variables and coefficients.
- Collapsing the distinction between SavedStates (which were used to initialize new games with variable values) and Turns (which were used to define the variable values for a given game turn) into the same single "State" object. The distinction between "starting states" and "active game states" is only semi-formal; implicitly, if a State is associated (by nullable foreign key) with a Game, then it is considered the state for a turn of that Game; if its *game* foreign key is null, it is a potential starting state. (For more on starting states, see the "course sections" documentation.)
- Storing the entire State object in an unstructured fashion, in a single text column, with its structure interpreted in Python code based on an independent Configuration schema. The previous system, by contrast, pushed data structure into the database, with separate tables for Variables, Coefficients, sets of Variables in a Turn, etc; and e.g. a separate row for each variable value in a given turn. This structure didn't really provide any advantages (we never had to run any complex relational queries or aggregations on variables and turns, with the possible exception of the graphing tool) and made it overly difficult to manage incremental changes to the logic engine's interface.

By pushing structure out of the database and into Python code, and by decoupling the definition of a structure from the data that fills it, incremental changes should be easier to make and keep track of. Similarly, by de-formalizing the distinction between variables and coefficients, the logic engine can be more flexible – it can decide what sort of interface it wants to provide, and no structural changes to the database will be necessary. (For example, configurations of "events" could be added; for details, see the "events" documentation.)

3.3 Future Directions

The design considers Configurations to be tightly coupled to the logic engine, and loosely coupled to everything else. Any change to the Configuration implies that the code of the logic engine has changed (specifically, that it will be looking for more or fewer variables or coefficients, or will be expecting a different type for the value of a given variable or coefficient) and, while not every code change in the logic engine requires a Configuration change, any code change that modifies the logic engine's interface expectations must be accompanied by a Configuration change.

In light of this, it's not clear to me whether Configurations should actually be stored in the database: perhaps, instead, a Configuration should be specified declaratively in Python code that lives alongside the logic engine. (The decision to store Configurations in the database was inherited from the previous incarnation of MVSim.) At the moment, notice that the choice of Configuration is hard-coded to the one whose ID=1.

The Configuration system, and the coupling of a Configuration to the logic engine, with Game and State objects therefore more-or-less agnostic to the data structure and logic engine that act upon them, opens up the possibility of multiple logic engines existing in a single MVSim installation. As long as each Game mediates between Configuration and States, it could dispatch to one of several logic engines. This could be used to provide:

- A/B testing of logic implementations
- Versioned logic engines with the ability to "roll-back" the default engine to a previous version

- Beta-testing new logic engine versions
- Providing different logic implementations to different courses, course sections, or institutions

Ultimately I'm imagining that the logic engine would be moved into a separate pure-Python package, which ships with a logic engine and its necessary Configuration schema; the MVSim Django application would then be a simulation platform, and one or more logic-engine packages could be installed and activated on top of it.

For more discussion of this see the "logic engine" documentation.

3.3.1 Object-Oriented Variables

The system of Configurations and Variables allows for composite variables that contain other variables, as sketched above. I've envisioned this being used to refactor and combine some of the existing variables, both to simplify some of the logic engine's code and to reduce the headaches on instructors and admins creating and editing starting states by providing logical groupings of conceptually related variables in the editing UI. (The UI benefits would occur automatically thanks to Deform's schema-to-form logic.)

In particular, the assorted list-like variables related to family members could be refactored into a dict-like Person variable with logical attributes, and then aggregated in a list-like People variable whose list entries must be Person items. These variables include:

- variables.names
- variables.genders
- variables.ages
- variables.health
- variables.education
- variables.sick
- variables.efforts
- variables.schooling_state

If these are combined into a Person variable, a bit of logic engine code would need to change: specifically, the marshall_people and setup_people functions could be removed entirely.

Other variables might also be good candidates for this sort of refactoring; symptoms to look for include code that transforms variables before passing them on to the meat of the logic engine (like the simple_controller.adjust_submission function in the *engine* module); variables with ad-hoc string formats that encode multiple pieces of information in a single field (like *sick*, *purchase_items* and *sell_items*) and variables which are rarely or never used in the logic engine on their own, but instead need other variables to be meaningful (like *bednet_ages* and *owned_items.bednet*, and maybe the various *microfinance_* variables)

3.3.2 Equations Map

Another future idea Rob and I have talked about is a way of browsing through the variables and coefficients in a configuration, and, ideally, seeing the equations that they're used in – essentially, converting the formulas in the PDF equations document to a set of database entries with each component being a hyperlink to the variable that it represents; and then a databrowse-style interface for exploring the relationships. Note that if the configurations and variables were converted to code this could be done just as easily in code.

CHAPTER 4

The User Input Model

The User Input model is like a "mini-configuration" that defines which variables may be specified directly by the end user in a given turn. This should be a strict subset of the variables in the active Configuration; the Configuration's variables that are *not* present in the active User Input (as well as all coefficients) are not directly changeable by the end user and are instead calculated within the logic engine.

Mostly you should read about the configuration system which will tell about how configuration schemas are defined and used.

The Colander deserialization of a user's submission should automatically invalidate user submissions that contain keys not present in the active User Input.

Just as it does with Configurations, the view code is currently hard-coded to look for the User Input with ID=1 in the database.

Note that the view code takes a game's current state, merges in the values provided from the form submission as deserialized through the active User Input schema, and then sends that merged state in to the logic engine for processing; the result of that processing is the only thing that is subsequently reserialized and stored in the database. (This is precisely the same approach that MVSim's TurboGears incarnation took.) In other words the user's submission (or the complete post-merge state) is never stored in the database as-is – and, since the logic engine is free to recalculate new values for variables that were directly entered by the user, the user's submission cannot reliably be reconstructed from the available data.

I've considered the idea of storing that post-merged-pre-processed state as well, though there's been no particular reason to do so; for alternate implementations of the simulation platform, though, this could become important – like gameplay with a staged set of decisions (e.g. "first submit the family-level decisions, then submit the village-level decisions") or a multi-player simulation.

CHAPTER 5

The State Model

The State model stores a single state of a game in a JSON formatted text blob, which is deserialized according to a Configuration.

Most of the important things about the state model are straightforward and/or covered in other documents (particularly the configuration system and the logic engine)

Note that the *created* timestamp of a state expresses the time when a user played a game turn that resulted in that state. This is actually the only data used to construct the sequence of turns that make up a Game's history, and to determine the current turn of an active Game.

5.1 Browsing and Editing States

For faculty and staff, special Django views are available for viewing, editing and cloning any given State. These views are intended to be used for developing starting states for students to use when initiating new games. (These are the named views "view_state" and "clone_state" in the primary URLconf.)

The "view_state" view is used for both viewing and editing a state. States associated with a game are not editable; but their data can be browsed, and they can be cloned into a new State that is not associated with any game and is therefore editable.

In these views, the forms displayed to the faculty user are dynamically generated using Deform[1], a framework-agnostic library standalone by Chris McDonough that is maintained as part of the Pylons Project. Deform is also used to validate submitted data for edit forms, and to display any error messages for invalid submissions. Deform works by using Colander schemas, so it's a natural fit for States, which are associated with Colander schemas through the Configuration system. For the most part it's straightforward and the code should be straightforward, but there were a few "gotchas" that might not be obvious and are worth highlighting:

1. Deform implicitly requires that every Colander SchemaNode in use (recursively) must possess a *name* attribute – this is used to build the HTML form inputs' name attributes. The Colander and Deform docs do mention this, but it's not enforced in the code (it would be hard to enforce without breaking the libraries' proper separation of concerns) and I missed it at first. If this constraint isn't met, data submitted through Deform will behave badly, and data might get lost on save; for (a bit) more see https://github.com/ccnmtl/mvsim/commit/8e28a07c6718b2aa8fce00002e483420fde846a1
2. Deform has a "read-only form" feature, which will use a separate set of templates to render a non-editable view on the data.[2] However, as illustrated by the sample read-only form[3] provided alongside the docs, this isn't really usable out of the box; its output isn't pretty at all. So, for the State objects' read-only forms, I circumvented this feature altogether and instead used standard read/write forms, but injected some Javascript into the response to disable all the form fields. My relevant commits are https://github.com/ccnmtl/mvsim/commit/b30807b3e488dd131b5367ca7e3f24748b96272c and https://github.com/ccnmtl/mvsim/commit/60b9232d2bbdf6dd4ccfa90bdf36203ea1d84321

3. Deform's default templates are built with Chameleon ZPT. Yes, that ZPT. Sorry.
4. Deform template overrides can be dropped into the *deform_templates* directory in the Django project – I customized one template to make it a little prettier for our usage. To override a form, just find the right one in the deform source distribution, copy it to this directory, and edit.
5. Deform treats HTML POST data as a stream, and thus requires that it be available on the server in precisely the same order that it was submitted from the client. This is a neat trick and actually a valid assumption based on the relevant specifications. But Django breaks this assumption – its HttpRequest.POST QueryDict is *not* properly ordered. It's necessary to work around this: see https://github.com/ccnmtl/mvsim/commit/4f9c1bdb4fff30129b49b4834ba070074baaad03#L1R27

[1] https://docs.pylonsproject.org/projects/deform/dev/

[2] https://docs.pylonsproject.org/projects/deform/dev/basics.html

[3] http://deformdemo.repoze.org/readonly_sequence_of_mappings/

5.2 Future Directions

The State model's main content is a JSON blob of all variables and coefficients, stored in a text field, and interpreted through the active configuration schema. This seems like it could be a natural fit for a NoSQL database – on the other hand, it's not clear whether the added complexity there would worth it. The only area where I can imagine any concrete gain is in the graphing tool, which has some really horrible and inefficient code for aggregating and averaging some variables into new graphable datasets; on the other hand, that code could pretty trivially be made much cleaner and more efficient without any database changes.

Relatedly, I was considering making States meaningfully versioned, with the version-dimension representing turns played, and extending the interface into the logic engine so that the engine would have access to the entire game history – which would let us eliminate some of the warts in the configuration like health_t1, health_t2 etc which technically track current values of past state. However, this would bring its own complications, since a single starting state would no longer be sufficient to describe a new game – so I'm now leaning against it.

CHAPTER 6

The Events System

The Events system determines what notifications to display to the user in the End Of Season Report, based on conditions expressed against the game's State prior to the turn being played and/or after the turn has been played. For simple, hopefully self-documenting examples, see the test cases at the bottom of the *engine/event.py* file.

Events include information like:

- Under what conditions should this event occur?
- What text should be displayed to the user if this event occurs?
- What is the event's severity? (Well, actually, "css class")

The actual Events in use are in a CSV file, *events.csv*, in the root of the Django project; they are read from that file as needed at runtime.

A Django setting, *MVSIM_EVENTS_CSV*, points to the location of this file by default.

6.1 Future Directions

Pretty straightforward.

- The events should probably live in the database, not a CSV file (which is an artifact of the Google Doc we were collaborating on to specify the events to build ou)
- The events should be associated with the Configuration somehow, I think, so that they can vary per game, and because they are technically a required component of the interface to the logic engine.
- The remaining notifications should be ported to the Events System: some are still just hard-coded in the Season Report templates.
- Exposing UIs for faculty to browse, modify and select the active events could be interesting.

CHAPTER 7

Course Sections

7.1 Courseaffils

MVSim uses the CCNMTL-developed Courseaffils[1] library to provide course-based user groupings for the entire user experience. From the user perspective, nearly all interactions with the application occur within the context of a single course; if the user is associated with more than one course in the system, he has to select which course to work through before interacting with the application.

A *COURSEAFFILS_COURSESTRING_MAPPER* Django setting is used to determine how logged-in users should be auto-associated with courses. By default it is set to a Columbia-specific backend that integrates with the WIND login system and Djangowind[2] auth backend.

The primary *courseaffils* model is *courseaffils.Course*. Courses must be added to the system via the Django Admin UI before users can be associated with courses. Course-student and course-faculty mappings can be added directly through manual creation of *auth.Groups* associated with the courses, or automatically by setting a Course's *coursestring*. (This automatic association feature is the bit that relies on the *COURSEAFFILS_COURSESTRING_MAPPER* setting and which defaults to a central-auth-based implementation specific to Columbia.)

On the backend, a user's currently active Course object is available as *request.course* which is set through a session key in the *courseaffils.middleware.CourseManagerMiddleware* middleware.

(A lot of the above documentation should be moved to Courseaffils, which currently lacks any high-level docs.)

[1] https://github.com/ccnmtl/django_courseaffils [2] https://github.com/ccnmtl/djangowind

7.2 Course Sections

Beneath the primary *Course* containers, users are further associated into Course Sections via the *mvsim.main.models.CourseSection* model. Course Sections determine only one thing: which Starting States are available for a user to start a new game from.

This currently lacks any useful UI and has to be set via the Django Admin UI.

Currently all students in a class are automatically stuffed into a single "default" CourseSection. The CourseSection is created via a *post_save* signal on *courseaffils.Course* in *mvsim.main.models* and users are added to it in ad-hoc *mvsim.main.views* code. No Starting State is auto-associated.

CourseSections could also impact views on the high score table, which isn't yet implemented anyway.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search