SAGE DevOps, Gameful Intelligent Tutoring, and Publication

Midterm Report

COMS 6901 – Projects in Computer Science, Fall 2018

Alex Dziena / ad3363

Lily Yu Li / yl4019

Contents

Abstract

Over the course of this semester, we are focusing on the DevOps MVP (Story #304) and Workstream Integration (Story #249) stories within SAGE Integration (Epic #339), the SAGE Feasibility Study & Publication (Story #340) within Survey, Field Study Design, and Publication Strategy (Epic #182), and the Intelligent Hinting MVP (Story #107) within Gameful Intelligent Tutoring (Epic #17).

In the first half of the semester we have worked primarily on 1) code standards, linting, and post-commit hooks 2) integration of prior GIT work into the SAGE UI 3) integrating the Parson's Puzzles work from the summer semester.

Architecture

Code Standards and Linting

We have selected widely used code standards for each primary language used in the SAGE codebase, and automated linting has been integrated into our continuous build processes. In almost all cases (with the exception of FlexPMD), we have selected actively maintained code standard libraries with rich toolsets. Adoption of code standards has been found to play a role in improving software project technical quality[1, pp. 355–371], and we believe that using widely adopted standards will provide several benefits over creating custom standards, including shortening the time to productivity, increased familiarity with researchers, and prior validation of the standard by a large set of academics and professionals.

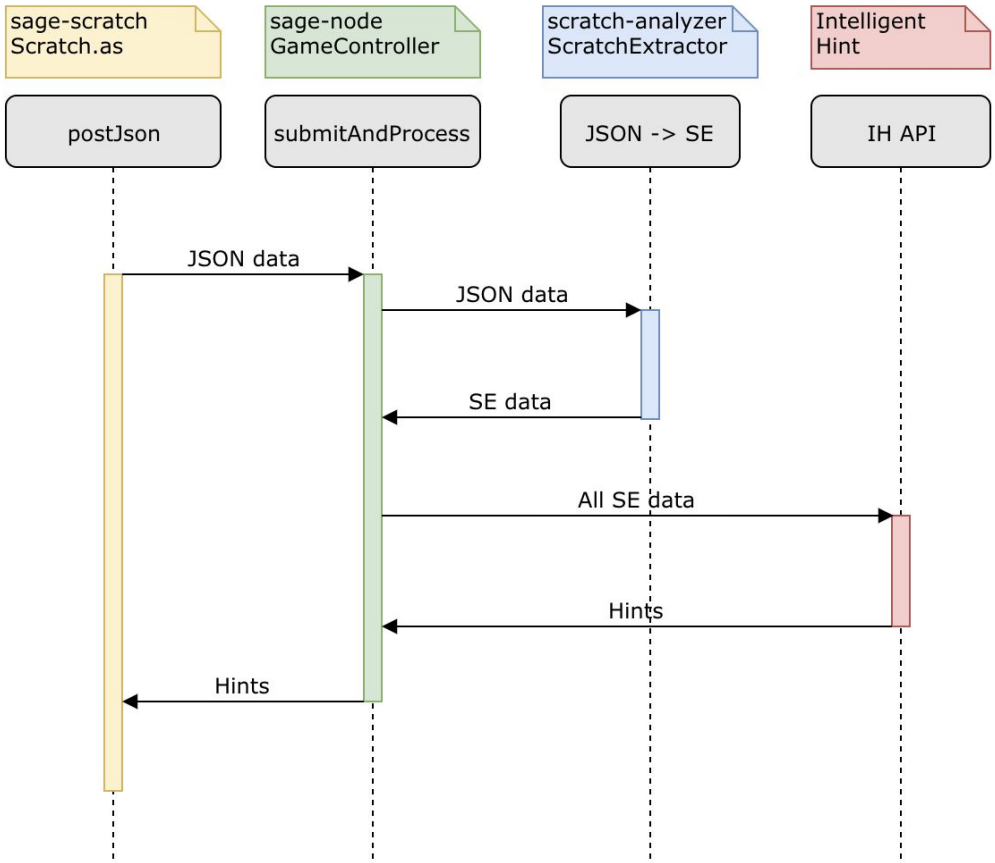
Post-commit hooks

We are using the existing TFS version control architecture to prevent direct pushes to mainline SAGE branches, and instead using pull requests: a commonly used mechanism to allow build and test validation before merging with a mainline branch. Although this mechanism currently requires manual acceptance of a pull request (after automatically running build and test validation), we are evaluating writing a polling system that polls for and automatically accepts non-breaking pull requests.

Sage-node integration

We will dispatch requests and serve data through sage-node so that we only need to maintain one interface for data exchange and one schema for each data collection.

Data flow diagram:



Sage-node will create a new JSON file and upload it to the database after receiving JSON data from sage-scratch. Then it will call ScratchExtractor to convert JSON data to SE data and upload a new SE file to the database. Finally, it will fetch all SE data associated with the game from the database and send them to the intelligent hint system to get the latest hints.

Parson's Puzzle integration

Merging work on Parson's Puzzles from the previous semester has proved very challenging. The approach we took involved cherry-picking individual commits onto a test branch, and testing integration for each commit. This allowed us to avoid bad/breaking commits in the sage-scratch repository, and resulted in a clean build currently available on dev.cu-sage.org.

Implementation

Code Standards and Linting

1. **Documentation**
   Documentation has been added to SAGE's wiki[2] detailing code standards for each of SAGE's primary programming languages. Documentation also includes guidelines for

introducing new style guides for newly adopted languages, and provides details on integrations with our CI / CD systems.

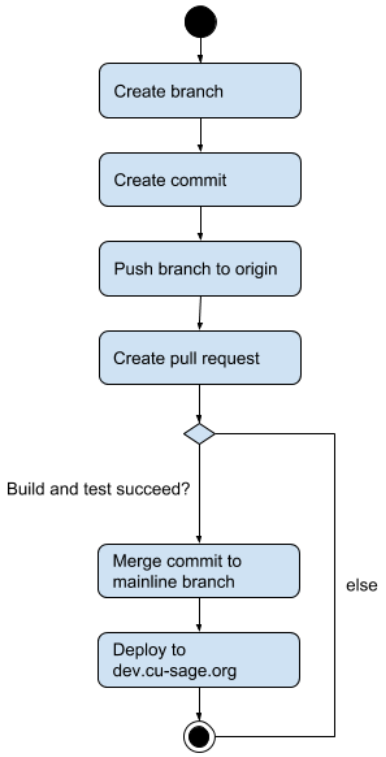
2. **Standards and Toolsets**
   We are using the following standards and toolsets:

Lanugage	Standard	Toolset
Actionscript	Apache's FlexSDK Coding Conventions[3]	FlexPMD
Java	Google Java Style Guide[4]	checkstyle
Javascript / Node.js	standard with semicolons[5]	semistandard
Python	PEP 8 style guide[6]	pylint
Ruby / Puppet	Puppet 5.0 Style Guide[7]	puppet-lint
Other Languages	Shellcheck Style Guide[8]	shellcheck

3. **Build system integration**
   Each linting / static analysis tool listed above will be run on every commit to all SAGE repos. At present, semistandard is integrated into the sage-frontend CI/CD system, which will be followed by FlexPMD, checkstyle, pylint, puppet-lint, and shellcheck (by order of priority).

Post-commit hooks

We are preventing breaking changes from entering the primary branches of all SAGE code bases (“development” and “master”) by preventing direct pushes to these branches. Instead, commits can only be merged to mainline branches via a pull request: a request is made in TFS by a researcher to merge a commit pushed to a non-mainline branch (e.g. a feature branch). This triggers an experimental merge and build, and if this build fails (including test failures) the pull request is rejected. Successful pull requests are built and deployed using our existing Continuous Integration and Deployment architecture.



Sage-node integration

1. **Run Java code in sage-node**
   Since we need ScratchExtractor to convert .json files to .se files, we want to run java code in sage-node. We implemented this with the node-java module.
   Previously, ScratchExtractor could only read and write from the local file system. We modified ScratchExtractor to support parsing a single JSON string and return a single SE string. The java project is exported as scratchJava.jar and added to the sage-node branch. The required files for the java project are also added in sage-node/FilesRequired.
   The java utilities are implemented in app/utils/javaUtils.js. Method extractJson takes a JSON string and returns an SE string.
2. **Store files in the database**
   To connect different code areas and get rid of the dependency on the local file system, we will store files in the database using MongoDB's GridFS. In addition to the file content, we can also store important information such as timestamp, student ID, game ID, and objective ID in the metadata. GridFS also allows us to fetch files with metadata. All utilities for file storage has been added to sage-node/app/utils/fileUtils.js. Right now, we have implemented postDbFile, getDbFile and deleteDbFile to directly upload, download and delete files from the database. Moreover, we added uploadJson, uploadSe and downloadSeFiles which return Promises so that the data flow can be created as a sequence.
   - uploadJson: upload JSON string to the database as a JSON file.
   - uploadSe: upload SE string to the database as an SE file.
   - downloadSeFiles: download all SE files with the matching metadata as a JSON array.

   We may also want to store hints and user types in the Game model.
3. **Hint request**
   Sage-node sends .se files to the intelligent hint system to request hints. The .se files will be sent as a JSON array returned by fileUtils.downloadSeFiles. Although the current intelligent hint system only generates automatic hints, the design should be able to support different hints types such as on-demand hints, chatbot hints, and multilayer hints.

   Request example:

```
{
   "seFiles:" [
   {
      "content": "<<Object Stage>>",
      "info": {
         "studentID": "stu1",
```

```
            "gameID": "game1",
            "objectiveID": "obj1"
             "timestamp": "21-12-28-GMT-0400-2017"
        }
    },
    {
        "content": "<<Object Stage>>\n\t\t\t<<Object Sprite1>>",
        "info": {
            "studentID": "stu1",
            "gameID": "game1",
            "objectiveID": "obj1"
             "timestamp": "21-12-30-GMT-0400-2017"
        }
    }
    ]
    "prevHints": [
        {“type”: “onDemand”
         “Hint”: <a block>
        },
       {“type”: “automatic”
        “Hint”: <a block>
       }
    ]
}
```


Depending on the intelligent hint system APIs, the request structure is subject to change.

4. **Hint timing**

   According to the hinting algorithm, the frequency of automatic hints is determined based on the user behavior/type. However, the current implementation (in Scratch.as) provides hints at a constant frequency (every 5 seconds). We want to hand the control of when to show automatic hints to the intelligent hint system. To achieve this, we may refresh hints every second in postJson, and when the intelligent hint system responds with an automatic hint, we show it to the user.

   Response example with explicit automatic hints:


```
[
    {“type”: “onDemand”
     “Hints”: [<a list of blocks in order>]
    }
    {“type”: “automatic”
     “Hint”: <a block>
    }
]
```


   But frequent updates will introduce a lot of redundant requests, expecially since it is unlikely that the JSON data changes every second. Therefore, we should only post

JSON data when there are possible changes (e.g. on mouse clicks). Furthermore, the intelligent hint system can include the timing information of the next automatic hint in stead of explicit automatic hint whenever it refreshes hints.
Response example with timestamp:

```
{“nextAutoHintTime”: <timestamp>
 “allHints”:
   [
      {“type”: “onDemand”
       “Hints”: [<a list of blocks in order>]
      }
   ]
}
```

When sage-scratch receives nextAutoHintTime, it resets the hint timer to show the hint at the updated time. Depending on how we generate on-demand hints, we may need a separate object to store automatic hints. For now, assuming that the on-demand hints are created using the same logic as automatic hints, we can simply pull the first hint from the on-demand hint list and serve it as the automatic hint.

5. **Show hints**
   After receiving hints from the intelligent hint system, sage-node passes them to sage-scratch. Sage-scratch then updates the hints on the BlockPalette and resets the auto hint timer (with nextAutoHintTime). When the timer goes off or the user asks for an on-demand hint, sage-scratch will pull the first hint from the on-demand hint list.

Limitations and Assumptions

Code Standards and Linting

FlexPMD is no longer supported by Adobe. We make the assumption that the final version of the tool will continue to function, and that SAGE’s actionscript version is compatible.

Post-commit hooks

Although a pull request triggers an experimental build, accepting the pull request still requires a manual step on TFS. We are investigating automating this so that all passing pull requests are automatically merged.

Sage-node integration

The current integration design assumes that the intelligent hint system and the user behavior detection system will be integrated together this semester and that they will share common APIs. For now, we do not send any user type information to the intelligent hint system, but this may change if the user behavior detection implementation requires us to store and send user type along with the SE data.

Parson's Puzzle integration

Work on Game Routes from the Spring 2018 semester had to be removed to successfully merge Parson's Puzzle work from the Summer 2018 semester. We will attempt to cherry-pick this work back onto the successfully merged mainline development branch later this semester.

Milestones and Future Work

In the second half of the semester, we will complete the remaining DevOps tasks, including Newman Configuration, improving test coverage, and implementing parameterized random testing[9] for the ML codebase, finish all remaining integration work for surfacing intelligent hinting in the UI, then shift focus to: 1) the Reference Architecture first draft, and 2) chatbot prototyping.

We've also defined two stretch goals. If time allows, we will focus on these following completion of our other goals. Alternatively, these could be addressed in future semesters

1. **Context-aware hinting mode selection:** Based on the path progression and wall time of a student in a given puzzle or constructionist game, switch the hinting system between "just-in-time" and "on-demand" hinting. Possibly, decrease the specificity of the hints based on inputs from the outer loop, i.e. how many assignments a student has completed weighted by the "dominant" mastery of each of those assignments.
2. **Chatbot semi-supervised training:** Use student outcomes resulting from a set of provided hints to train the chatbot hint provider in real time. Improve the pedagogical efficacy of hints by retraining the hint provider on a batch basis, potentially using question/hint/outcome triples from non-SAGE corpora.

Milestone	Date	Status
First draft of code standards completed	2018-10-05	Complete
Coding standards complete, Post-commit hooks completed	2018-10-12	Complete
Newman Configuration	2018-10-19	ToDo
Reference Architecture outline completed	2018-10-26	In Progress
Complete integration of prior GIT work into SAGE UI	2018-11-02	In Progress
Improve test coverage for all repositories to 30%, including AsUnit test suite for sage-scratch and parameterized random testing for ML codebase.	2018-11-09	In Progress
Continued work on chatbot prototype	2018-11-16	ToDo

Chatbot prototype, test coverage at 60%	2018-11-23	ToDo
Chatbot UI integration	2018-11-30	ToDo
Reference Architecture diagrams / data visualizations completed	2018-12-07	ToDo
Reference Architecture first draft completed	2018-12-14	ToDo

Sage-node integration Implementation progress

Milestone	Date	Status
Implemented Java utilities to run java in node	2018-11-02	Complete
Updated ScratchExtractor	2018-11-02	Complete
Implemented GridFS file storage utilities	2018-11-09	Complete
Added new routes for file upload/download testing	2018-11-09	Complete
Data flow: upload JSON, generate SE. and send hint requests	2018-11-09	Complete
Update postJson frequency in Scratch.as	2018-11-16	ToDo
Update Scratch.as to show automatic and on-demand hints	2018-11-16	ToDo
Data flow: process responses from the intelligent hint system and send hints to sage-scratch	TBD	Blocked by GIT

Citations

[1] L. M. Maruping, X. Zhang, and V. Venkatesh, “Role of collective ownership and coding standards in coordinating expertise in software project teams,” *European Journal of Information Systems*, vol. 18, no. 4, pp. 355–371, Aug. 2009.

[2] “Confluence.” [Online]. Available: https://gudangdaya.atlassian.net/wiki/spaces/SAGE/pages/579141633/SAGE+Code+Standards. [Accessed: 09-Nov-2018]

[3] “Flex SDK / Wiki / Coding Conventions.” [Online]. Available: https://sourceforge.net/adobe/flexsdk/wiki/Coding%20Conventions/. [Accessed: 09-Nov-2018]

[4] “Google Java Style Guide.” [Online]. Available: http://google.github.io/styleguide/javaguide.html. [Accessed: 10-Nov-2018]

[5] *standard*. Github [Online]. Available: https://github.com/standard/standard. [Accessed: 10-Nov-2018]

[6] G. van Rossum, B. Warsaw, and N. Coghlan, “PEP 8: style guide for Python code,” *Python. org*, 2001 [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[7] “The Puppet Language Style Guide,” *Puppet*. [Online]. Available:

https://puppet.com/docs/puppet/5.0/style_guide.html. [Accessed: 10-Nov-2018]
[8] V. Holen, *shellcheck*. Github [Online]. Available: https://github.com/koalaman/shellcheck. [Accessed: 10-Nov-2018]
[9] C. Murphy, G. Kaiser, and M. Arias, “Parameterizing random test data according to equivalence classes,” in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007, pp. 38–41.