Introduction

This document starts with a brief description of the system and main features and later it goes into excruciating detail.

System Concept

- Language translations are done from text that often is presented in some kind of context. The context is either visual or associates with previous text, etc...
- Most tools currently allow text translation without presenting context
- When context isn't available ambiguities cannot be resolved accurately
- Translations can happen automatically or manually, this system allows users to review them
- There are 3 main roles:
  - Reviewer
  - Manager which is typically an LPM (Linguistic Program Manager) and
  - Producer which is the person uploading content to be reviewed.
- The system presents 3 central items:
  - An Application: This is the same conceptually as an Android app, but could also be a web app, or anything else that has screens.
  - A Screen: This is a screenshot image and its associated data. For Android screens map roughly to activities. For anything else is just the simple concept of a screenshot.
  - A Review: This is the main unit of work from a reviewer. It is either OK, or has issues.
- Each language has a locale. Locales are associated with screens and reviews.
- There are two types of screens:
  - Base screen which is associated with only the base locale. A base locale is typically English
  - Screen, always associated with a base locale and with its language locale.
- A Round which roughly maps to the concept of a project and the round is used to identify the current work being done for an app. Only one round can be active and the round number increases in numerical sequence.
- There are two main parts of the system:
  - A back-end that handles all business logic, and
  - A front-end that handles the presentation and user interaction.
- The system displays screenshots for two locales one of which is a base locale. The reviewer can then verify the correct translation in-context.

Use Cases

This section only describes the main cases

Logon

- A user logs on to the system, through the web interface.
- The concept of a logon only applies to the front-end, because this is how OAuth2 works. This process is very simple for our system because the complex logic is handled elsewhere.
- Upon successful logon the user receives a token which is subsequently used for interacting with the back-end.

Administration

- Ok, this is not a use case but more of a collection but the concepts are very similar.
- Applications (android apps, for instance) can be added or changed. Change is discouraged, but it currently does most of the right things, but it's quite elaborate. Typically, an app would be added by the upload process, which populates the database as needed.
- Administration should mostly deal with users and apps. Administration of screens is best left for the system or the Producer, and this process is described below.
- TBD:

Upload

- This process can also be understood as data generation, update, screen management, etc...
- Notice that although upload is conceptually different from data generation, a future goal, is to have this process happen in one step.
- It is at this stage where the producer can specify or override application parameters. These are for instance, currentRound or encryption enforcement.
- Data is generated in at least two ways that are supported currently
  - Android screenshot generation, this is done through an an Android app that allows a Java developer to write scripts that generate screenshots and data. This app may actually be replaced by another system that automates this process, freeing us from having to write the scripts
  - Screenshot image import, this is a utility that allows a producer to import screens. This utility allows a producer to upload screenshots from a directory or directories and it infers where the images will go based on file and/or directory names.
- The screen generation creates a CSV file that we upload to the system through a web interface. This process is manual, but is a single step, the old system had a multi-step upload process, that was much more complicated.
- There is a way to cause the android device to upload the file directly, I got about half way done through this. But it basically works by having the user logon to the app. The problem is that we don't have an 'actual' android app. The screen generation app is actually code that is run by the Android UIAutomator SDK system utility. So I need to figure out a way to allow a proper logon through this app. Once this is possible, the android screen generator would be able to send the file directly to the back-end. It turns out that this is a feature people seem to think very useful, because it enhances security and makes the system easier to operate.
- When the file upload complete, the system will:
  - Populate database, if there is new data. In this context new data means screens that don't exist for the round being uploaded. Think of screen data as a place holder, there isn't really anything more to that. So a new screen record or entity is only generated for a new round. Screenshot images however, are a different story, these can be updates, and may replace existing ones. This process of replacing is best left for the manual validation process done by the producer.
  - Extract screenshot images and copy them to GCS (Google Cloud Service.) The front end does not have direct access to these files. The access is handled securely through the URL Fetch service. When an application is set to be encrypted, the server will use a token sent during the upload process, not the password itself, and will encrypt the image bits before copying to the cloud. We don't encrypt the whole file, only the RGBA content, this allows us to know if the file becomes corrupted for whatever reason, but more importantly, the server does not care about what image the file contains, only that the right image file is associated with the right screen and sent to the user.
  - Process the images in a most likely Borg process that will attempt to discover duplicate images. The duplication discovery is what they used to refer as fingerprinting process, I don't like the term because it does not allude very well to what it does. We can find a better name or just call it effort deduplication. Basically, the process tries to find images that are similar enough so that the reviewer can be made aware through a screen status and visual cues. We can handle that in at least two ways: 1) Automatic, the system will enter a review record stating what the duplicate image and screen has so as to make them equal in status. 2) Manual, the reviewer will have to review again but will see a cue that will indicate that this work has been previously done for the same screen and different round or for a different screen, by the same user or a different user.
- When the upload process completes, the screens are not yet visible to the reviewer. However, it seems that it may be beneficial (TODO: discuss) to auto-approve screens in cases that do not overwrite others. In other words, for new uploads, not updates. For the screens to become available the next step must happen.

Validation

- This step must be done by the Producer, and it should typically be done right after the upload
- One goal is to make the system as flexible as possible while keeping complexity at a minimum.
- The producer will do a visual inspection of the recently uploaded screen. The visuals are almost exactly the same as what a reviewer sees, except that instead of seeing the issue submittal modules, the producer would see statistics relating to (TBD) possibly missing screens, etc...
- The producer will have the option to compare the screen with previous version, via a standard list box.
- The producer will have the option to discard a screenshot or to 'approve' it. This approval will make the screen visible to a reviewer. An approval can override a previous screen in case it may have previously existed.
- The validation should not take too long to do. Care should be taken so insure that screens are the correct ones for the reviewer to work on.
- Notice that this whole process, upload and validation do not actually alter any screen data, it is merely the screenshot images that are changed. So the process does not have any risk of clobbering data regardless of possible mistakes.
- In the event of an upload error, or even a user error, previous screens are never deleted, they are always associated with: app, base locale, locale, round, and version. It is the version field that allows us to have multiple screens for the same round. The version field does not exist in the database, it is simply a sequential number assigned to screens when they previously exist, when the back-end process copies the files to GCS. Furthermore, file versions are only visible to the producer. For example, after an upload, if a screenshot existed for say round 1, and we are submitting for the same round, but the file changed, the new name would be something like myfilenameetc_1.png, while the previous one would just be myfilenameetc.png.

- NOTE: the Borg background process would attempt to find images with high similarity, and the threshold will be configurable, in order to prevent effort duplication.

Review

- A reviewer would visually inspect the two screenshots for a base and language locale and will verify correct translation. Or in the case of incorrect translation or other issues, the reviewer will be empowered to notify the pertinent parties.
- In the reviewer inspection process a new review will be sent to the application indicating success or failure. In the case of failure the review will contain data that the system will use to perform the necessary notifications.
- Upon submission of a review the system will send back a review record with possibly extra information that will be displayed in the reviewer user interface.

Navigation

- The system will allow navigation to other apps, locales, screens, statistics or other parts of the system, including management functions when applicable.
- TBD:

Security

- A user must be authorized for each operation. This is currently accomplished in 3 ways:
  - Morpheus: if a user exists in Morpheus it is automatically permitted access, unless explicitly revoked via override.
  - System: this is the functionality that the user admin module will provide when ready, and it's explained in detail in the security section.
  - Override: this allows the initial deployment to proceed without the admin module, but it also has other interesting incidental features that may become useful.
- The system has security features that allows: Authentication using OAuth2, and Role Based Access Control (RBAC) in two forms, Mandatory Access Control (MAC) and Discretionary Access Control (DAC). They are considered best practices.
  - OAuth2, all operations must be authenticated and resolved to a valid user, through the standard mechanism mentioned above in Logon.
  - MAC deals with authorization based on user roles. Users are assigned roles by their managers. The operation/role authorization mappings happen during system setup. At the moment they are 'hard-coded' but the system can be easily adapted to make this process dynamic (at run-time.) However, currently the system security features appear adequate, because of the next feature, DAC.
  - DAC deals with authorization based on the concept of a 'resource.' A resource is either one or both, an app, a locale. This allows a manager to authorize a user for only a subset of the required review work, or even for submissions (uploads.)
  - The system will also filter any results returned to the user so that only appropriate information is sent back.
- Additionally, an app can be 'encrypted' which means that screenshot images will be encrypted with a key that only the user would know, and the user must enter it during review. If this password key is lost or forgotten the system should (for security reasons) have no way of recovery.
- Incomplete list of possible security issues:
  - Because of system flexibility, there is a lot of overlap, and this may be a good thing or a requirement. DAC authorizations are inclusive, in the sense that the system will attempt to find an authorization source. The implication is that, currently as designed, you cannot 'prohibit' a user a resource. They are prohibited by default, until any manager gives authorization. However, this would not be too difficult to extend due to the modular design.
  - For encrypted apps, the server does not save the password or even the token, all is discarded after the upload. So the system cannot be hacked, unless a hacker actually is able to decrypt the file, and this is understood to be very time consuming, in the order of many years, until quantum computing creates a problem for us. If the password is lost or forgotten, the user better have a backup of the image files.
  - The files are decrypted in the web front-end, when the user enters the password. The password is only saved for the current session and the system will ask for the password every time. This is so that the sensitive information has less risk of being compromised, at the expense of usability. There is always that tradeoff.
  - The system features caching of authorization outcomes for all operations and types and users. The cache window is currently set to 1/2 hour, but is configurable. And, it works as follows:
    - A logged on user performs any operation. Something gets cached.
    - The user keeps using the system and performs operations at intervals less than 1/2 hour.
    - The user takes a lunch break for two hours and comes back to use the system.
    - The cache is expired and cleaned
    - The user took too long for lunch, his manager is upset and fires him remotely and revokes his credentials. Let's say for example that his LPM does this in Morpheus, but for some reason is unable to communicate with the user
    - The LPM was unaware of the 'purge user' feature and does not invoke that functionality.
    - As long as the user continues to use the system at intervals less than 1/2 hour, he will still be able to use the system.
    - A solution to this is to run a cache flush operation in the server at regular intervals, say every 2 hours
  - The system allows the proliferation of users with management roles, this may not be an issue at all, but care should be taken during system administration. The way this works is that a manager can make other users managers. However, even managers are subjected to DAC authorization rules. Which basically means that a manager is only a manager for those resources that he/she has been authorized. For instance, take the clock Android application, LPM_1 manages it, and makes Reviewer_1 a manager. However, LPM_1 also manages the settings Android app. In the current system design, this means that Reviewer_1 now also manages the settings app. This is currently the default behavior and it was done to facilitate administration. However, once we have the admin module ready, in the next iteration. When a user is made a manager, the manager who authorized this would assign only those resources which he/she wishes to share. But, notice that the now manager Reviewer_1, because he is a 'proper' manager, he can now make other users, managers. So again, this may not be an issue, depending on the work-flow. Security is a pretty tricky business, and these kinds of details usually have to be worked out somehow.

Feature/Benefit (Partial List)

Features	Benefits
Removal of legacy schema	Performance, Comprehensibility
Data centric using Google Data Store, replacing database centric mysql approach	Performance, Scalability, Reliability, Usability
Switch Android centricity to Application centricity	Usability, Extend use to web apps. Note that system will still capture android data for future use and data mining
Discard the concept of product in the processing. Product information is available in Morpheus, Discard the concept of script	Comprehensibility. Note that product and script information will still be captured
Integration with Morpheus. Application, Product and User information, Role based Authorization	Synergy with existing systems, Avoid effort duplication
Removal of xml manifest processing, which imposes labor intensive work-flows for developers and users alike	Performance, Comprehensibility, Usability
OAuth2 Authentication and Role Based Authorization, Encryption of images for sensitive applications	Security
Separation of back-end (Java) from front-end (Dart, JavaScript), API based interface (Google Cloud Endpoints)	Performance, Scalability, Comprehensibility, Security, Easier Mobile Device Support
Google Technology Stack, Object Oriented, Modularity, Caching, Validation,	Performance, Reliability, Comprehensibility, Security

Differences with Old AViK System

- ISSUES:
  - System as provided was broken. There was a discrepancy in the schema between the 'Console' and the 'Reviewer' web apps. The problem was that the newly created database for a new product had some fields missing. At this point we can no longer trust the system because of the likelyhood of other more subtle discrepancies
  - The source code for the 'Workbench' was not made available to us, limiting our ability to trouble shoot the system, or to extend it
  - The system was incomplete and several key parts of the system were untested. The fingerprinting feature was not tested
  - The system had a legacy tables that were not used, but only partial functionality was removed so the system operation still populated some tables without actually needing to
  - It used java libraries from 2008 that were no longer in active development, and there was a risk that in the future they may not be compatible
  - The web code base was, very far from being able to pass code review
  - The system would have been difficult to extend, for example, in order to add security functionality, there is not central point of access, because of the page centric paradigm used
  - Initially we were unaware of the schema incompatibility and the effort was focused on refactoring
  - One of the reasons the workbench was necessary was that there is a bug in the Android system where the sqlite file does not properly save image files (blobs) I've provided a work around by converting the image binary content into base64 which is ASCII and is supported in the blob
- The new web based system, front-end and back-end is a complete rewrite, code reuse was less than 1%

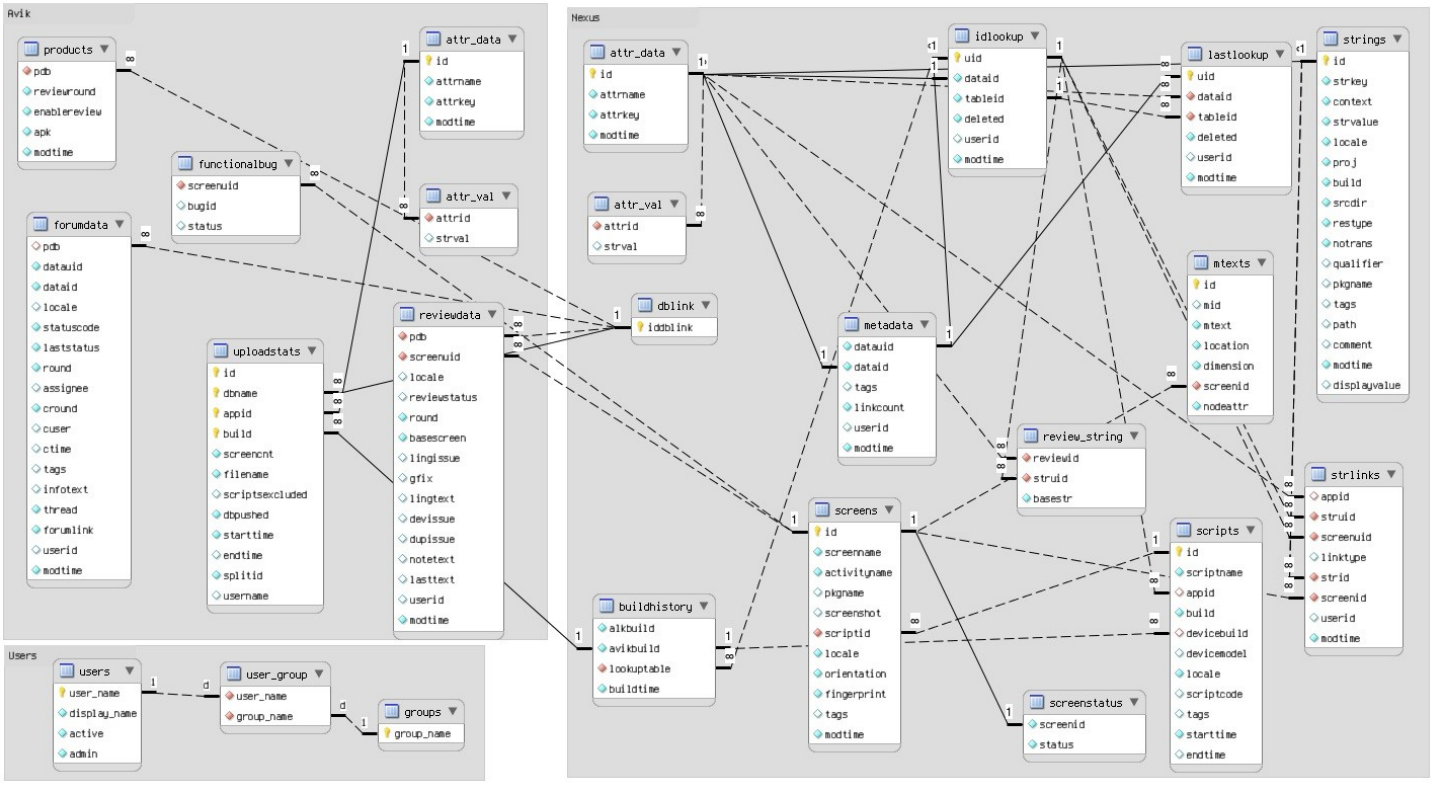
Original AViK	New System
Android centric approach, limits usefulness	Supports Android, but can be used independently
High complexity in schema	Minimalistic schema, leverages Google Cloud DataStore
Requires producers to create databases	Only one DataStore, system is database agnostic, and users never deal with them
	This concept is not necessary for system operation, however data is still captured because it

Requires management of product (an Android release and version)	may be necessary for reports
Requires management of rounds, not only creating, but starting and stopping	Rounds are also required but they are dealt with during upload and validation more simply
Concept of a 'Manifest.' Complex management of xml files by producer requiring them to enter all 'permitted' screens. This would have prevented the use of a 'crawler' or similar Android app that creates content without scripting. In essence it ties the web app to the Android screen generator limiting the system usefulness	Discarded. The responsibility falls on the producer to create valid content in the validation stage, which was also the case in the old system. Simplified producer work-flow
Antiquated web application style, web page centric with complex navigation	Modern design, desktop client feel, as Gmail for example. Simplified navigation based on presentation of data, not page navigation, using standard list boxes. Reduces server trips and greatly increases performance and enhances user experience
Caching not provided	Caching supported at both back and fron ends. Greatly enhanced user experience
Lack of security, system does authenticate a user but the enforcement is hard-coded in application configuration files. If a change is required you needed to edit these files and redeploy the application	Security Module uses best practices. Role Based Access Control RBAC with Mandatory Access Control MAC, and Discretionary Access Control DAC. Please see below for details

Entity Relationship Diagrams: Old and New Systems

Original AViK Schema

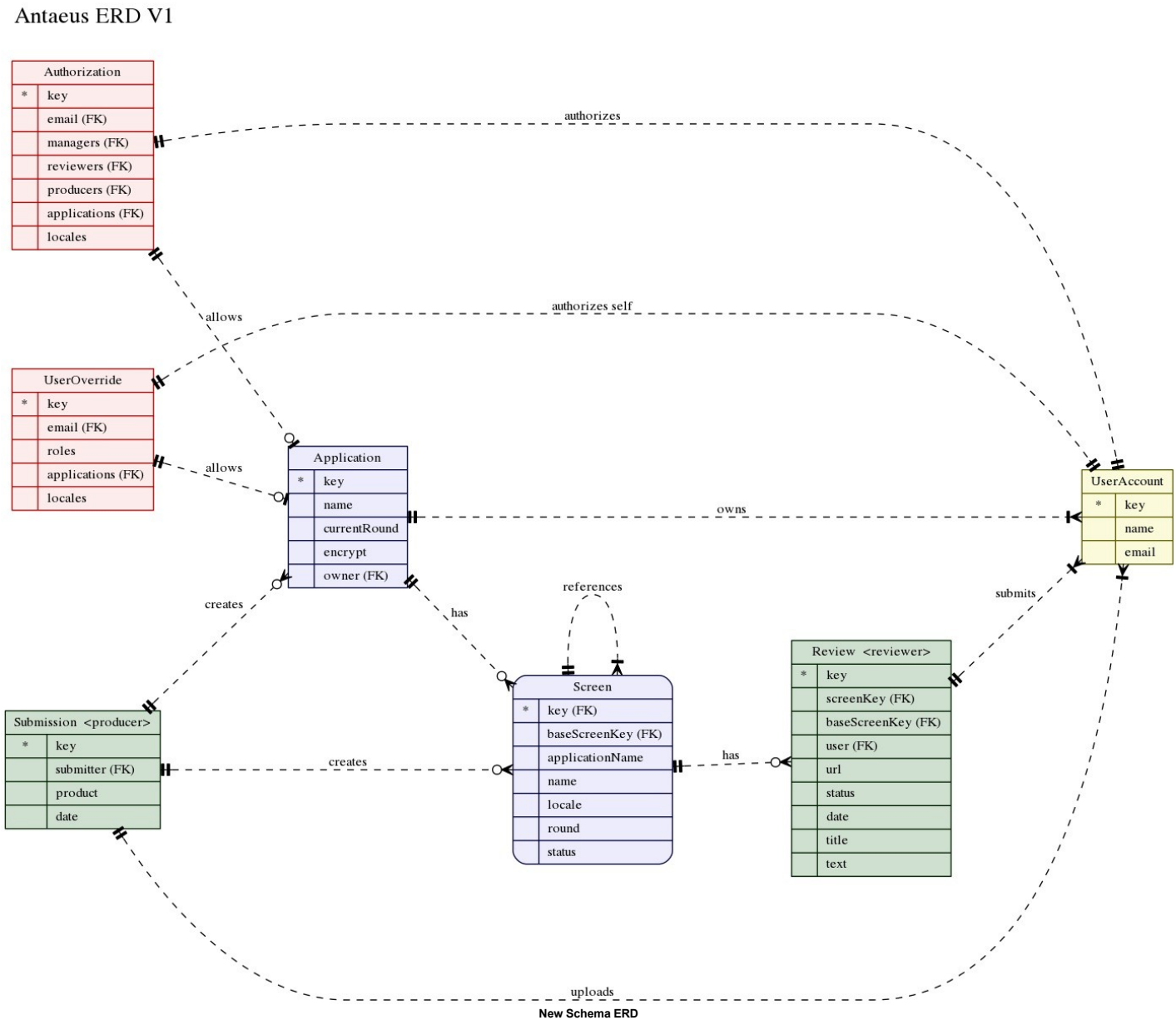
- 3 actual SQL databases, though Users db was not used. Two databases were the minimum operation but it has the concept of a 'master' database and the secondary database is what the producer must create for different products or other reasons. Brittle and poorly performant database design.
- Below in the diagram you can see db Avik which is the master db, while Nexus is an instance of the secondary database, and during system operation there would be many of the latter. This would have the side effect of making backup operations difficult to manage.
- Many tables were not actually used, the forum system for instance. The string related tables had optional use but was not tested.
- Uses Entity Attribute Model controversial database technique, completely unwarranted for such a conceptually simple system.
- Wikipedia EAV
  - Entity–attribute–value model (EAV) is a data model to describe entities where the number of attributes (properties, parameters) that can be used to describe them is potentially vast, but the number that will actually apply to a given entity is relatively modest. In mathematics, this model is known as a sparse matrix.
  - There are certain cases where an EAV schematic is an optimal approach to data modeling for a problem domain. However, in many cases where data can be modeled in statically relational terms an EAV based approach is an anti-pattern which can lead to longer development times, poor use of database resources and more complex queries when compared to a relationally-modeled data schema.
  - In the words of Prof. Dr. Daniel Masys (formerly Chair of Vanderbilt University's Medical Informatics Department), the challenges of working with EAV stem from the fact that in an EAV database, the "physical schema" (the way data are stored) is radically different from the "logical schema" – the way users, and many software applications such as statistics packages, regard it, i.e., as conventional rows and columns for individual classes. EAV tables conceptually mix apples, oranges, grapefruit and chop suey
  - Scenarios that are appropriate for EAV modeling: 1) Modeling sparse attributes, 2) Modeling numerous classes with very few instances per class: highly dynamic schemas
- Stack-overflow: EAV is notoriously problematic as it leads to severe deployment performance and scalability problems. pros-cons, EAV in sql
- SQL Central: I have long had a dislike of Entity-Attribute-Value models within RDBMS's based on my experience trouble-shooting and supporting them in various forms over my career. The premise of such a model is that they offer huge flexibility as objects that would usually require extensive physical modeling can be represented simply by records within a generic schema. My experience is that the promised flexibility of such models is illusive and more than offset by the penalties and inconveniences they incur. With the rise of NOSQL solutions, native handling of XML within SQL Server and polyglot data solutions the limited number of appropriate use cases is dwindling still further. A colleague of mine was studying for a masters degree in BI. One of the professors described EAV designs as write-only. Obviously such a description is an exaggeration but the comment was made to prove the general point about the difficulty of getting data out of an EAV design.
- WordPress: The EAV pattern looks very appealing at first glance. The upsides of EAV are usually most apparent during design and development. The downsides don't tend to manifest themselves until after the application has gone live.
- Proper use in bio informatics: EAV



Original Schema

The new schema is simpler

- Application, Screen, Review and Submission have been explained.
- The UserAccount entity is simply an entry that contains information about the caller. Remember, that even though this is a back-end, all operations must be authorized, and that each call carries with it a token given to us by the OAuth2 system. The back-end, more specifically the Google Endpoints library translates this token somehow and it converts it into a appengine User object and then we use that to obtain account information from Google+.
- The two red entities deal with security and they basically are somewhat analogous.
  - Authorization deals with managers
  - UserOverride applies to an individual user
  - These two entities are actually 'authorizers' that the security module indirectly uses to query for authorization and resource information.



New Schema ERD

How does it work?

NOTICE!: some features are either: TBD, to be developed; WIP, work in progress or need testing

- Back-end system provides RESTful API using Google Endpoints.
- Front-end system provides web based user interface using the Google Dart language.
- The security module authenticates and coordinates authorization for the user, operation and resources. This work is also cached for a predefined and configurable time window.
- Data and screen images are fed through the import/upload procedure and database entities are created.
- Data is stored in Google DataStore.
- Screenshot images are stored in Google Cloud Services (GCS.)
- After the upload, the producer will do a cursory verification and validation of screenshot images, and will approve or reject them individually.
- The reviewer will then be able to see the images and screen data to submit reviews.
- The system caches data in two ways. The back-end caches data from the data store and the front-end caches image data and other data.
- When the system is asked to do some work through a Google endpoint API call, the security check and input validation happen alternating in several steps for better efficiency, so as to prevent unnecessary system load.
- The operation is then performed. Some operations generate events that are handled immediately after, or queued in a task.
- After event handling, some operations return data as a list of entities, a single entity or a result. What is returned to the caller is referred to as product. This product is subjected to a security filter before sending.
- The front end then receives the product and will display the data and perform any subsequent operation.

Technology Showcase:

- Google OAuth2 user authentication
- I10n Team Security Module with RBAC, OAuth2, MAC & DAC, plus image encryption
- Seamless integration with I10n Morpheus and Query Manager.
- Java Object Oriented Modular and light weight design. The Eclipse IDE is used for development.
- Dart Object Oriented Modular design for the web based back-end, with HTML5 support.
- Google AppEngine back-end, featuring:
  - DataStore support through JDO
  - Google Endpoints, provide REST API entry points
  - Memcache, for caching common data
  - Modules: Mail, Users, Task Queues, Channels, URL Fetch, OAuth2
- A Google Dart tool is also used to produce a library that the web front end uses to call the back-end.
- Google Borg Cloud Computing Platform, for CPU intensive image processing (TBD)
- Google DataStore based on Cloud Big Table for state of the art scalability
- Google Cloud Storage, GCS, for screenshot images file storage, only directly accessible to the system.
- External libraries:
  - Lombok, creates getter/setter, constructors and other boiler plate code, saves time and aids in system comprehension
  - Joda Time module for date/time handling, considered best practice, the use of Java date/time classes is very strongly discouraged

    - Jackson JSON parser, for interaction with external systems, such as Morpheus
    - OpenCSV for importing screenshots and screen data from android devices
- External dependencies and libraries kept to a minimum, all listed above.
- JUnit for unit tests, plus tests for: integration, stress, load and concurrency
- JavaDoc and Doxygen with GraphViz/Dot and MSCGen are used to generate documentation and diagrams.

System Architecture:

- Google Technology stack for scalability
- Single Access Point SAP security, considered a best practice
- Object Oriented modular design, use of software design patterns, self-documenting code
- Dependency injection for better testability, extensibility and ease of maintenance
- Factory classes and methods as appropriate to separate system initialization from operation, this also aids in system comprehension
- Non-relational database schema design
- Light weight API interface with approximately 10 general application (non-administrative) functions, minimizing server trips, for better performance
- CRUD (Create, Retrieve, Update, Delete, and also List) operations are handled uniformly using Java generics and the strategy pattern.
- The strategy pattern is also used for all other API operations including events, so that the same mechanism is used. This is done in the controller.
- The main logical (conceptual) modules are:
    - API, the classes here provide all API entry-points (Google endpoints)
    - Security, there are two, one is a general purpose security module that is system agnostic, and two, the part that understands system roles, operations and resources.
    - Cache
    - Collaboration, integration with Morpheus, QueryManager, etc...
    - Model, DataStore Entity definitions
    - Storage, handles persistence of entities in a data source agnostic way
    - Services, UrlFetch, Upload, System Start-up/Shutdown, Task Queuing, JSON, etc...
    - BLL, Business Logic Layer, explained in more detail below...
- Great care was taken to avoid code duplication so as to make the system easy to extend and maintain.
- In order to make the system unit testable, it was necessary to decouple most of the system components, increasing the design complexity and the code base by at least twice. Nevertheless, considering that this system does a lot more than the original one, the code base is still less than half and a great deal less for library dependencies.
- Input validation done consistently on all inputs
- Uniform error and exception handling with consistent reporting to appengine system logs

- The Iterable interface is used throughout and it sort of ended up taking center stage

```
Iterable<UserOverride> overrides = (Iterable<UserOverride>) relay.getStorage().query(email);
for (UserOverride override: overrides) {
  if (override.getEmail().equals(email)) {
    relay.delete(override);
  }
}
```

System Concept

System Initialization: to build objects and make system ready for use. The objects are built to specification because in the integration tests the objects might be built differently according to how we want to test the system.

- Upon system start-up, the factory creates a controller object, built to specification. Therein, the security controller creation is delegated to the security module, but parameters are passed so that the security module is aware of operations.
- During the creation of the security controller, objects that provide query capabilities and other resolutions are passed in. The concept here is that the security module does not know or understand what needs to be authorized. It only knows who to ask, and these are the objects that are passed in. In this manner, security is modular and we in fact have 3 sources of authorization providers,
    - System Authorization, this is an object that saves data in entities that are only served per manager. The entity contains lists of: managers, reviewers and producers, and locales and apps. This entity allows a manager to assign management roles to other users, in essence, they will then be proper managers.
    - System Override, this is an object that saves data in entities per user, and allows us to disable/enable a user, regardless of what the managers configure, so great care must be taken not to missuse this feature. There is also a purge user feature. The entity or record contains lists of: apps and locales.
    - Morpheus, provides the following, locale information and Lpm and Reviewer information.
    - Note that security is a very complicated subject and even more complicated to implement. There were not simpler mechanisms available, so we had to 'roll' our own.
    - The above objects can be thought of as 'authorizers.' More details will be provided in the security section.
- The system uses Dependency Injection in two of its forms, system initialization during object construction, and run time through method invocation. I tested the Google Guice framework for dependency injection and even though it is actually very nice, because of the way the system is designed with Factory methods and Builders, the dependency injection is best handled there, so no external library was necessary for the use of the technique
- The factory methods and builders construct the controller and the security module to specification, depending on needs and for what purpose, for example we can build one system for testing

System Operation: Object interaction is complex. Several steps must happen before the actual operation can be carried out. We want to use a divide and conquer approach and use separation of concerns, yet at some point entities either interact or are acted upon in group. Prior to the operation, validation, security and other steps are performed. CRUD operations are handled separately, commands and events are also, however, for non-crud API calls, we converge in an object called a Mediator. At this point, we have all the data we need from the caller, or has been retrieved. Here, we are in a state of bliss where everything is possible and we don't have to worry about bad input or security. This way, the API call can be carried out without further delay.

Preparation:

- A web app user (or another system) invokes an API entry point.
- The endpoint (same as entry-point) creates a parameter object and invokes a controller
- The controller creates a product object and coordinates several actors
    - The auditor/clerk maps input to operation, that is, maps the operation to an entity handler.
    - The sentinel authenticates the call, that is, checks the token given to us by the OAuth2 system, and,
    - authorizes the operation based on the user role, this is the MAC part of RBAC. The security model already contains all valid operations and applicable roles, it only has to query the user role from the 'authorizers'
    - The controller creates a relay object. This object contains the Mediator and all entity persistent handlers, plus context which is parameter and product.
    - The clerk resolves input, that is, insures that all required input is available for the particular operation.
    - The inspector (input validator) validates input based on pattern matching
    - The sentinel authorizes the user to the requested resource, this is the DAC part of RBAC. At this point we know what the user is trying to do and what app and locale would be affected. Now the security module will query the authorizers to see if any manager has granted any type of access to said resource.

Command Operation and Event Handling (Strategies):

- The controller will obtain a reference to the object which is to run the command. However, remember that the actual code could run in the mediator or elsewhere.
- The operation is then performed by the appropriate object.
- The controller will attempt to obtain a reference to an object which might handle the event. The event being the command we just ran.
- The event is then performed by the appropriate object.

Complete call:

- The sentinel will inspect the product and query the authorizers to see who allows the passage of the resource back to the caller.
- In the event that any part of the code should throw an exception, or an error code could be signaled. A detailed error message will be logged to the appengine system log for subsequent analysis and resolution if applicable.
- The control returns from the controller back to the endpoint.
- The endpoint will simply return the result or re-package the object if it's a collection. The object returned back to the caller very closely resembles the original entity. This is considered by the Google Endpoint and AppEngine teams, and by many others, a best practice.

For most operations, what is returned to the user is a result code which should be the string "OK." Otherwise, an error message would be returned and the message should be 'sanitized' to prevent information leakage. In the case of administrative operations, the web app would simply operate on the object/entity, edit as appropriate and return it to the back-end as a simple CRUD operation for updating the system. This mechanism, greatly simplifies the system and allows to deal with editing of data in an abstract form. This helps keep the code base to a bare minimum also.

Security Module Background

The security module is based on Role Based Access Control

- The use of RBAC to manage user privileges within a single system or application is widely accepted as a best practice
- Applications including Oracle DBMS, PostgreSQL 8.1, SAP R/3, ISIS Papyrus, FusionForge, Wikipedia, Microsoft Lync, Microsoft Active Directory, Microsoft SQL Server and operating systems implementing SELinux (Linux, Solaris and some other Unix-like operating systems), grsecurity (Linux), TrustedBSD (FreeBSD), and many others effectively implement some form of RBAC
- Wikipedia: Role-based access control
- OWASP:
    - Guide to Authorization
    - Access Control Cheat Sheet
- NIST: National Institute for Standards and Technology,
    - Intro
    - RBAC
    - FAQ

Security Module Features

This module uses some of the techniques that are part of RBAC as applicable, because we didn't need all the functionality

- Security Patterns:
    - Single Access Point (SAP), Initially implemented as a single method call to the module. Currently implemented as 3 method calls that alternate between security and input validation to increase performance.
    - Check Point, Conceptually similar to above, both happen in the Controller and only there.
    - Limited View, Operations that return data are also subjected to the check point, where a filter removes any unauthorized resources.
    - Model View Controller MVC, The security module is driven by the application Controller, and it is also a Controller that drives the OAuth2, MAC, DAC and Cache sub-modules.
    - Authorization Cache, because of how the module works by querying data, although DataStore reads are very fast, we still want to minimize that access. During a successful authorization its context is hashed and cached in a hash-map. The system has the capability of flushing the cache on request, at intervals or at particular events.
- Implements RBAC using MAC and DAC. The module supports specific types of access to a resource such as Read/Write/Owner, but it is not currently use because our usage patterns do not need it
- RBAC Model:
    - U = User or Security Principal is the email address
    - R = Role = Owner/Manager, Producer, Reviewer, (all other Unauthorized)
    - P = Permissions = role based permission to an activity, identity based permission to data
    - S = Session = A mapping involving URP, (we call this Context, because the system is stateless and a session might implicate state, there really is no session because operations are sort of atomic, so context)
    - A user can have multiple roles.
    - A role can have multiple users.
    - A role can have many permissions.
    - A permission can be assigned to many roles.
    - An operation can be assigned many permissions.

    - A permission can be assigned to many operations.
- Time based cached authorizations, with configurable parameter
- Limitations:
    - Denial of Service Attacks DOS, Google's AppEngine has a feature where you can add an IP address to be denied service. This, does not offer a reactive way to handle a DOS, and also, there is a limit as to how many entries the black list can have
    - The module currently does not handle DOS attacks, however, we could add functionality that could partially alleviate this shortcoming. A way to do this would be through keeping a list of IP addresses that have been serviced, and each call we would log the time, and if we see suspicious activity the code would 'react' by dropping the operation. We should also do not log for all, since that in itself could cause a DOS symptom.
    - Managers are trusted only for their owned resources, but they become owners when granted access, so the possibility of missuse is there, but that isn't a system issue.

Security Module Concept

The main concept revolves around a module that knows:

- Abstract: Roles, Access, Operation, Resource. Abstract means simply that the module does not define what they are, the implementer (application) will be responsible for knowing what they actually mean.
- What questions to ask, via Query Interfaces that implementers must know how to answer
- Authorizers, these are objects that are given to the module by the application so that the module can ask questions, termed queries
- Security sub-modules that know how to delegate the questions asked, via the authorizers
- How to resolve security questions asked by the application controller, by delegating to the right sub-modules
- The sub-modules implement their own cache, since they do the 'real' work, well not strictly true because, remember that all the modules do is to delegate the questions back to the application
- Why go through all this, and not just do the 'real' work in the application? That would defeat the purpose of modularity and system comprehension would be more difficult. So the module sort of 'forces' the implementers to do the 'right thing' to obtain the benefits of a best practices security module

The implementer knows:

- Concrete: Roles, Access (not used yet, but implemented) Operation and Resource
    - Access, Roles and a ResourceType, are implemented using non-mutable data, explained below
    - Operations are a simple Java enum, however Operations have attributes, that describe non security related features used by the validator and other parts of the system
    - Resource, this is a bit more complicated but not too bad for our application, see below
- How to manage authorizers which are part of the system. The system currently has only 3 authorizers:
    - Authorizer: (sorry about the name,) this is basically a DataStore entity and its associated container manager, that answers questions for a particular manager, it can answer for associated persons (Managers, Reviewers and Producers) and associated resources (Applications and Locales)
    - UserOverride: like the above, but only answers for one user, and the above mentioned resources
    - Morpheus: Morpheus is not a DataStore entity like the above, but a collaboration 'partner' so to speak, that implements almost the same functionality as the above. It can answer for persons (LPMs, considered managers, and Reviewers) and the only resource it knows for our purpose is Locales
- a Mediator, this is a class object that handles the actual method calls from above at a converging point, and knows to trigger system events when appropriate, such as cache flushing

The module and implementer do the above by:

- Module is application agnostic, the application or security implementer must derive from a group of classes, implement several interfaces and manage its own data persistence
- Is lightweight, less then 500 lines of code, the module does not manage any data other than the cache
- It 'consumes' data by querying 'providers.' A provider is a class object that can answer questions from the module, the questions are 'query' Java interfaces, for example getRole()
- A factory method in the application invokes the Security Module factory method which as parameters must receive: Operations/Role map,
- The Roles and Access class objects have a few interesting qualities:
    - Immutable properties, the only mutator is a private constructor
    - Immutable groups, these objects cannot be added or deleted, they are set when the system initializes, that is when the Java VM is started by the AppEngine system in the cloud
    - They were designed that way because they are security tokens and must be tamper proof. This comes at a price, we use a weird syntax when using them in the code. However their use must also be as flexible as mutable objects, because a user can have different roles at different places (authorizers) and we must compound the roles. This is accomplished by having a function that returns a reference to the right object. Example: An upload operation is attempted, Morpheus tells us that the user is a Reviewer, so Role role = Role.Reviewer, so we must keep looking, later an authorizer Authorization object tells us that it's ok, the user is also a Producer, so we do, role = role.getAdded(Producer), now the role object is a reference to Roles.ReviewerAndProducer, but the actual object referenced did not change. This seems a bit ugly, and in this application we aren't really going to sabotage our security and cause the reference to change for all others that share the same reference.
    - The point is that you cannot change the object even if you try, unless you can hack the Java VM, which is not impossible, just very difficult. The need for this use case is more evident when you are running code from libraries, or if you are implementing a library, or must interact at the code or object level with other systems.
    - Ok, I get it, but, why not just allocate the objects as needed? That way, we don't have to worry about changing references and the silly syntax and what not. Java memory allocation and garbage collection have come a long way since its early days. I think allocating memory unnecessarily is really ugly and should be avoided. Just think how many times during the run time the system has to allocate all these little critters. It would give the GC a good workout.
- Resources: The system deals with two items that for our purpose are considered resources. Applications and Locales. Therefore, a user can be given authorization to them via DAC. While MAC deals with Operations/Roles, DAC deals with User/Resources, thus we provide a complete solution to the security authorization problem. These are the details:
    - Resource implementers derive from Resource in the security module
    - Something must know how to handle the resources, in other words, how to determine if a user is authorized to it. Of course it is the Authorizer object that knows how, sort of...
    - The resources have an in-built factory method that knows not only how to build them for also how to manage them. Again, here we want to be as efficient as possible, and this is accomplished through cache or actually more specifically an object pool that is managed by the Resource class
    - The first time a resource is built it is saved in a hash map, so in essence they are singleton objects. I know, singletons are bad, but only when their scope is external. In our case they are managed internally through private functions that do various operations about them such as comparison.
    - There are 3 types of resource classes for the 2 types of resources:
        - EmptyResource: Uh!, this is a bit of a hack, it's only purpose is to allow us to add a new resource, but because of DAC wants to authorize everything and we don't want to handle special cases in special ways, this class allows a user to, add an application for instance, but, because the application does not yet exist and DAC will still have to compare it with something, this class allows that type of comparison. The premise is that when the call comes in for an Application Create Operation, it carries with it the new Resource "ApplicationName", an object of this type is built, but a search is done first to see if the user is authorized to the non-existent resource, some place in the logic sees this and creates this class object, which causes the authorization not to fail. Keep in mind that this happens after MAC, and that only authorized users such as Managers, or producers, during upload, are allowed to create the precious resource type Application. There are a couple of other use cases for this. I won't bore you with those details, yet.
        - ApplicationResource: This class only understands Application names, and does not care about locales. This enables some operations to be authorized in the context where the locale is absent. For example, the SetCurrentRound web API call.
        - ApplicationLocaleResource: This constitutes a pair of an Application name and a Locale name
    - The resources implement the **Comparable** Java interface, but this is of limited use to us, so they also implement a function called isLike() which can do polymorphic comparison to the other resource types. The logic is encapsulated in each resource type.
    - The 3 resource types form a hierarchy: ApplicationLocaleResource -> ApplicationResource -> EmptyResource -> Resource -> Resource (in Security Module)
- The queries are answered with Tri-boolean logic (or is it Trilean?) When a questions is asked an authorizer is not required to answer No or Yes, it may simply not be responsible for such resource or operation, for that case the answer will be Unspecified. This would leave room for other authorizers to be able to answer. An authorizer may answer in the negative at which point the query ends and the operation dimmed unauthorized and the user notified.
- Do not be deceived by the apparent system complexity, security is supposed to be very complex, in fact the implementation is quite light weight,
    - about 400 LOC for the 'Module'
    - about 800 LOC for the implementers
    - about 400 LOC for the DataStore Entity models, Authorization and UserOverride
    - and about 300 LOC for Operations and system related logic
    - all about less than 2K LOC, plus the unit test code

I attempted to reuse security functionality from external libraries but they would have taken also time to learn and implement in code, did not offer all the required functionality, did not like their performance characteristics, it is also hard to trust security code that is embedded in a library since it's more 'obscure' or at least harder to look at. The one that came close was Apache's Shiro which I thought was very nice, but it is sort of tailored for 'web apps' that act more like the typical web app, functions by using Java String objects that describes different things, like operations, resources, etc... and I thought this is not as performant and harder to manage, plus, we still needed to create the objects that must interact with it, so overall, the benefit that we could have gained would have been, in my estimation, much less than what we gained by rolling our own. Shiro is somewhat lightweight, about 1/2 a megabyte for the complete binary distribution. The source code for the core distribution is almost 12k LOC, and almost 5k LOC for the web support. The library is more geared toward EJB (Enterprise Java Beans) and I wanted something that plays nice with AppEngine, which there was none.

In Summary, the Security Feature, comprised of the Security Module and the Application Security Implementation:

- has:

    - 2 Controllers: Application and Security, the former manages the order, the later the who
    - Actors: User, Authorizers (Authorizer, UserOverride, Morpheus) and Enquirers (OAuth2, MAC, DAC)
    - Roles: Manager, Reviewer, Producer
    - Access: (not used currently) Read, Write, Owner
    - Resources: Application, Locale
- OAuth2 authentication is handled by Google, external to us, but we check for validity only and trust that the token is authentic and current
- MAC authorization is setup at system init via passing a map of Operations/Roles
- DAC is Authorizer dependent and is dynamic in nature depending on existing data and what happens to it during run time.
- is a kind of wrapper that marshalls Q&A between Enquirers (MAC/DAC) and/or Authorizers (DAC), about Operations (MAC/DAC) and Resources (DAC)

Security Module Use Cases (how the system uses it)

This section only describes the main cases in simple terms.

Authentication

- There is no concept of logon for a web API because this happens in the front end
- The front end receives a token which is handled by the Google Endpoints library and sent to the back-end
- The back-end receives the token and again Google Endpoints AppEngine portion handles the decoding and token validation and converts it into something we can use, such as a User class object. This object contains the user's email address and it is this which we use from now on to identify a user

API Call

- By the time the back-end receives the call, Authentication has already happened
- As mentioned earlier, the next steps happen in an alternating way with the validation process
- These processes are explained below, individually as use cases, even though they happen in the same API call

Authorize (first call by the system controller)

- The first step handles OAuth2 verification, currently is very simple and it only checks that the User class object is valid, ie. not null. Later, we can also check this against the database, but in fact this step already happens during DAC
- The second step handles MAC. This operation triggers a query to search for the user Role for the Operation, and only returns when this Role is sufficient for it, or if it is denied by an authorizer, it ends there.
- Because the actual query operation must ask the question to the authorizers and this is typically a DataStore query operation
- Note that the DataStore caches only some type of data to Google's AppEngine MemCache, because memcache is global is basically anybody can read it, so we don't cache there any security related data) it caches if successfully authorized this fact (not the data) to a hash map. The hash-map is secure because it has no data, only the hash number and it is in the Java VM application memory and never elsewhere, plus this is internal data to the security module and there are no accessors for it.

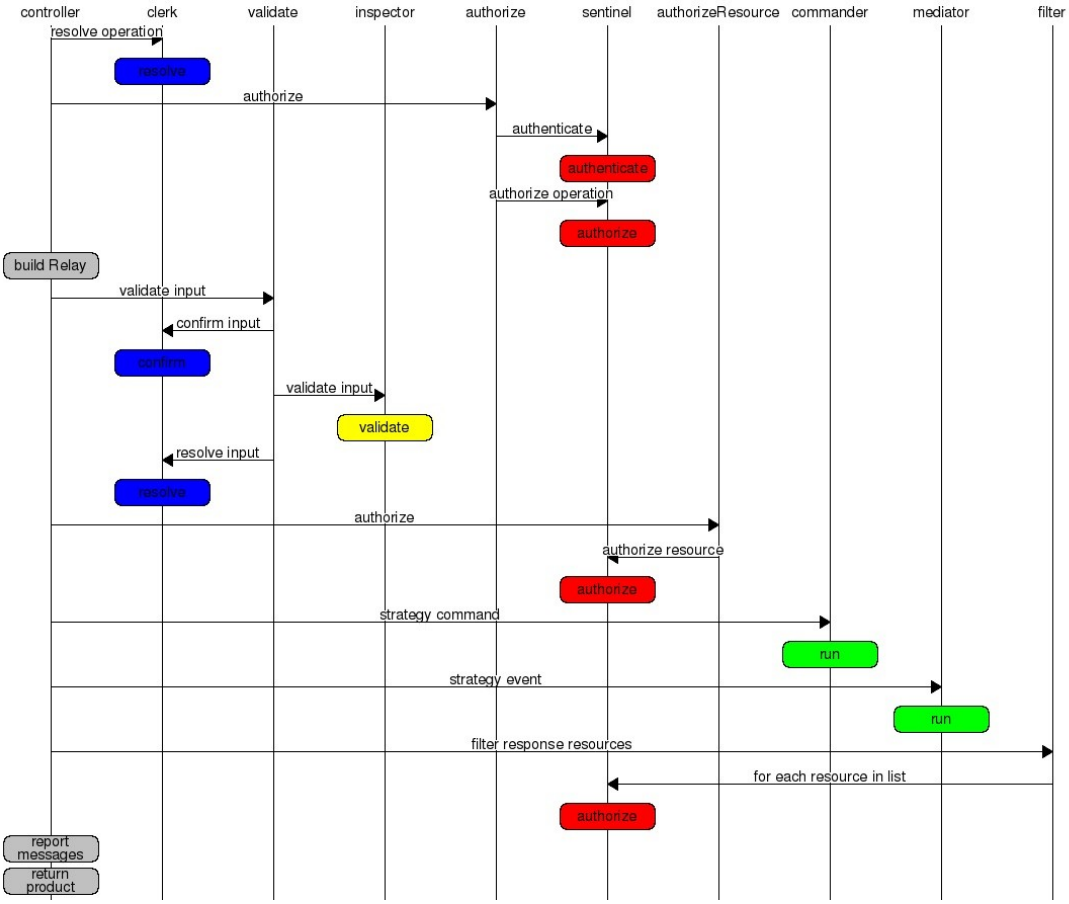
Authorize (second call)

- Because we now need to authorize a resource to the user, the controller builds a Resource class object from the API call passed in parameters. Remember that the Resource we build depends on what's available to us, so this resource can be Empty, Application or ApplicationLocale.
- The security module now invokes the DAC functionality (Enquirer) for us. This module is the only one that knows the system Authorizers (Authorizer, UserOverride and Morpheus)
- They are invoked sequentially until a yes or no response is given, or until exhausted at which point we take it as a negative response and deny access
- The system authorizers then get to work by querying the data store as follows: (except Morpheus which does it differently)
    - They all implement interface AuthorizeResource
    - Authorizer: causes the data-store to query one entity, if this succeeds, the user is in fact a manager, only managers own these types of entities.
        - The user is a manager, but a manager of what? The entity contains lists of applications and locales, that are authorized for ALL users that this manager manages.
        - Currently this may be a limitation, and if necessary we could extend the system to contain Resource pairs individually applicable. This is not a big deal at all and these changes would be localized in the Security Mediator, but currently we don't seem to need it.
        - The entity's users are 3 lists for: Managers, Reviewers and Producers. Whoever is found in the manager's list is a proper manager, because that user will have his/her own associated entity.
        - If the user is not found, we must query all entities, this would not be that time consuming because, how many managers do we really expect to have? The Authorizer code in the Mediator loops through all records until it finds that a manager has authorized this user and that this manager actually is authorized to manage the resource in question.
    - UserOverride: The user override as the name implies sort of does away with the manager's wishes and it basically answers for only one user, so no need to loop.
        - When a record of this entity is found for the user in question we can see the Role there right away, so, no need to extract that information via list membership as above
        - Note that we can block a user by setting his/her role to zero
        - The absence of a record for a user just means that the user does not have any overrides and only the Authorizer and Morpheus speak on the user's behalf
        - The record also contains the authorized resources for this particular user
    - Morpheus: These operations are more expensive because this is a collaboration module that implements the above mentioned interface, but must obtain its data from the Morpheus system by making a more expensive web API call to it. This is not that big of a deal either.
        - Morpheus data is precomputed at intervals since the data is fairly static and we really don't want to bother the system too much
        - This data, again is not cached to memcache for security reasons, but if given the ok, can be. We simply have lists of Lpms and Reviewers.
        - Morpheus can also give us information about Locale assignments which we use as authorizations for its users
        - Notice that this module has limited capabilities for Resource resolution, because, even though Morpheus has extensive information about 'Applications' the definition of that there in is broader. An Application to Morpheus means any Google application, such as Gmail, Android itself is another Application, but it is not currently clear weather it knows about specific Android apps, such as 'clock' or 'settings' which are the kind we would typically review for
- The DAC module caches authorization hashes for the user, and the hash maps the Operation/Resource pair, so that there is no confusion or security risk related to this

Filter

- Some operations return data, however the DataStore query makes no distinction about any type of authorization issues
- The system controller has a method that acts as a filter, by authorizing each Resource individually. Remember that these operations are cached so they happen quite fast

The above use cases can be illustrated as follows (it includes the input validation steps, since they are usually considered part of a security strategy):



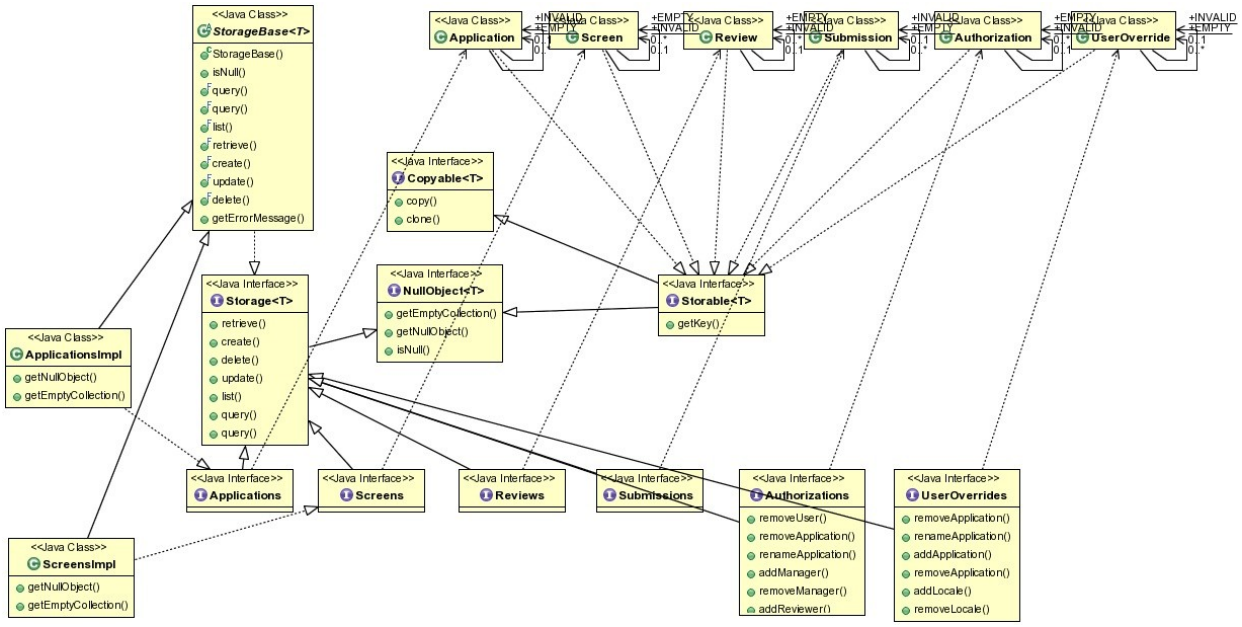
.

Model Diagram - ERD -> ORM

The following diagram shows the DataStore model (without fields or functions for brevity):

- The Model classes
- The DataStore handler, reponsible for each model type
- The Interfaces for dealing with them, for instance the DataStore classes
- It's a partial view of the system, where you can see only two implementer on the left, Submissions, ScreenDetails were omitted

DataStore Model Diagram



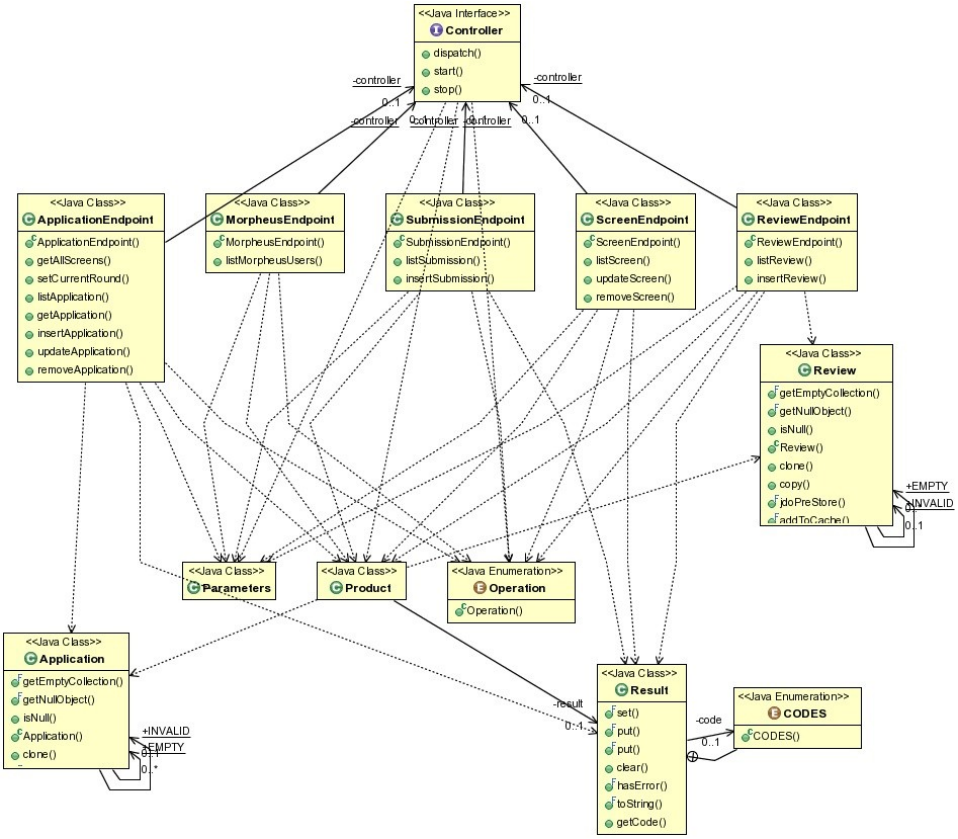
System Model

Controller and Strategies

Diagrams for a view to the controller

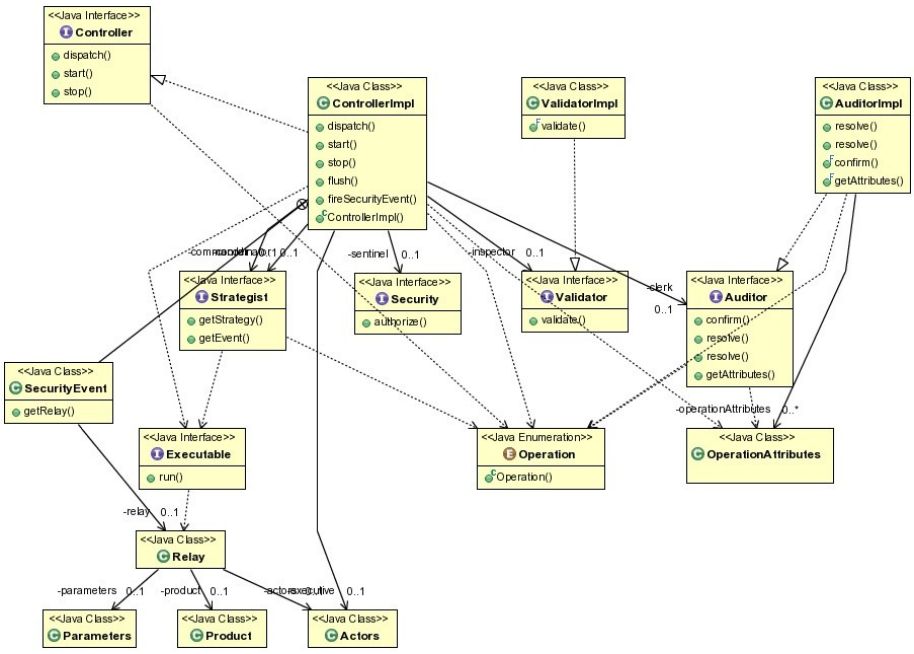
- Endpoint
- Main
- Command Strategy classes
- CRUDL Strategy classes

Endpoints Diagram



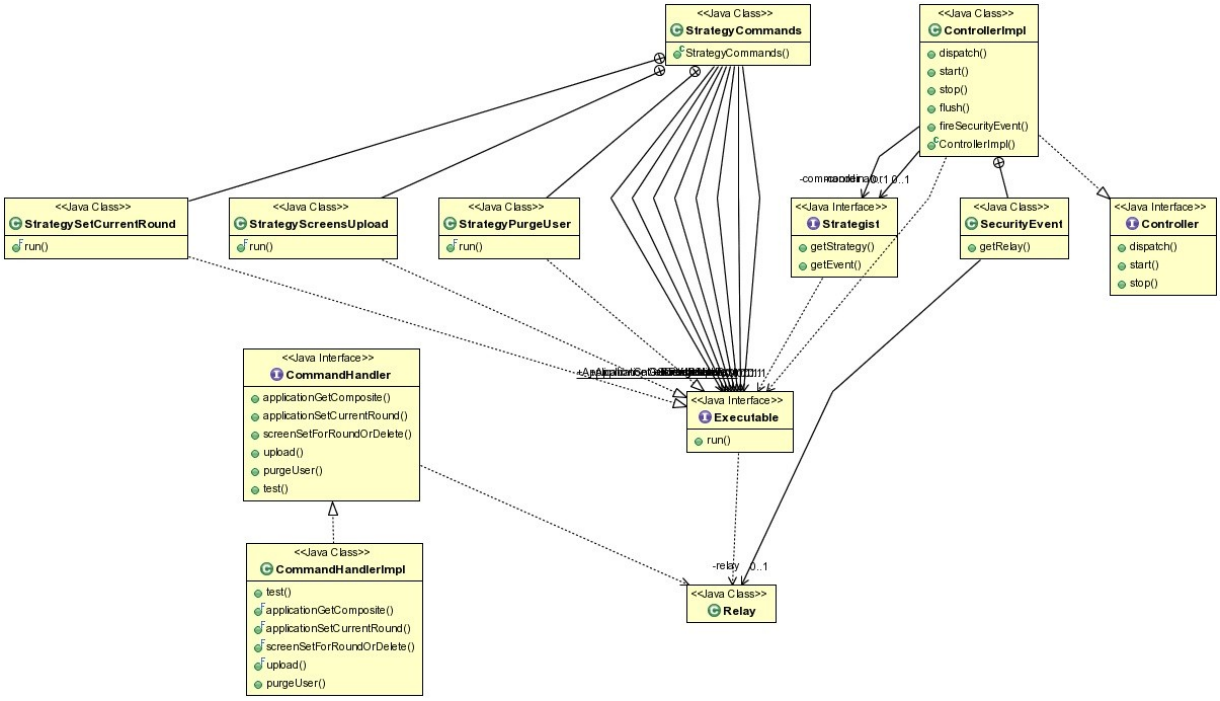
Endpoints

Controller Diagram



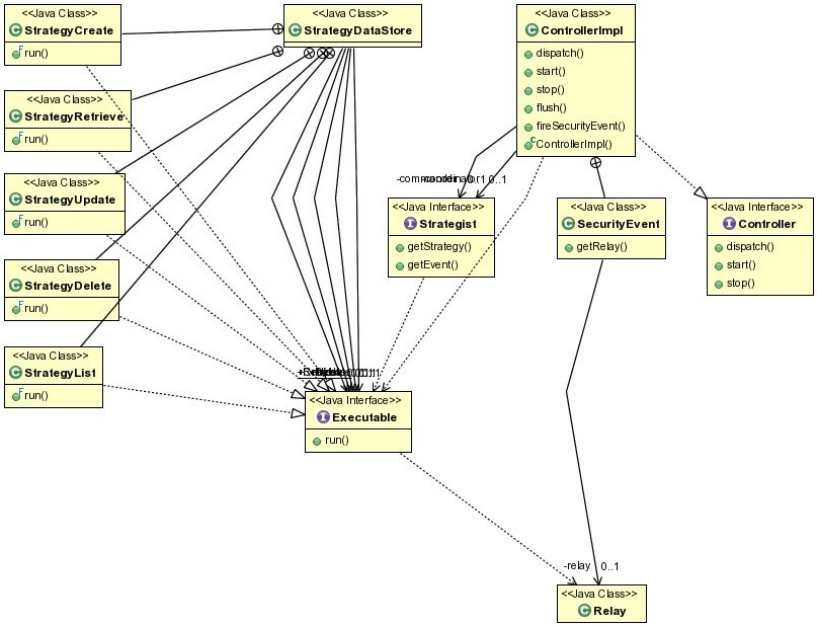
Controller

Strategy Diagram



Strategy

CRUDL Strategy Diagram

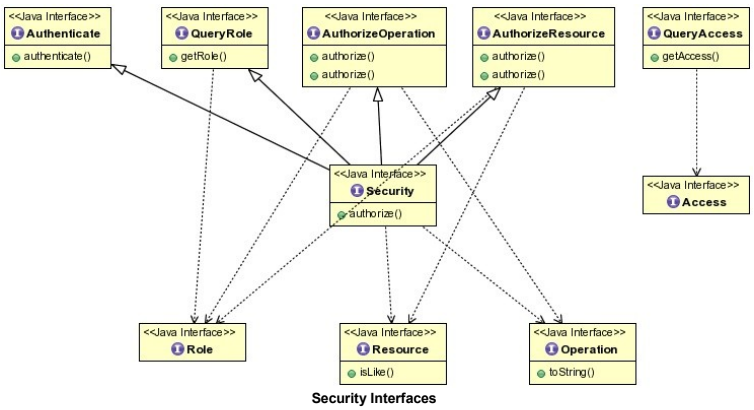


CRUDL

Security

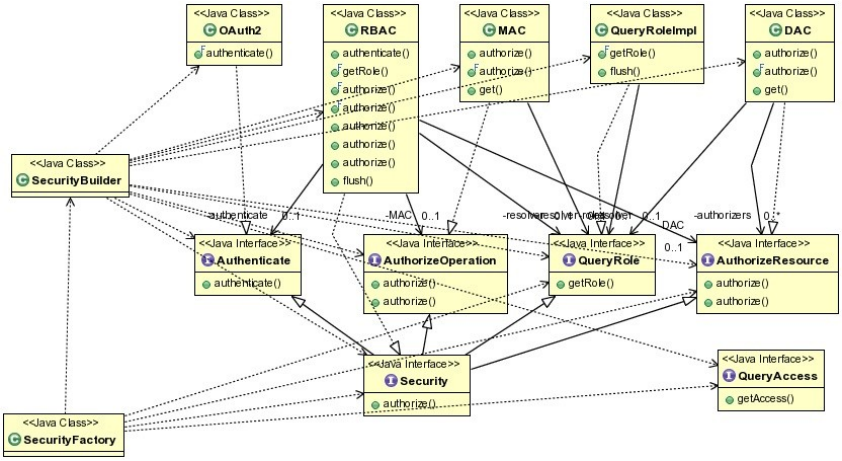
Diagrams for a view to the security module and related classes

- Interfaces
- Module
- Cache
- Application Security Interfaces Diagram



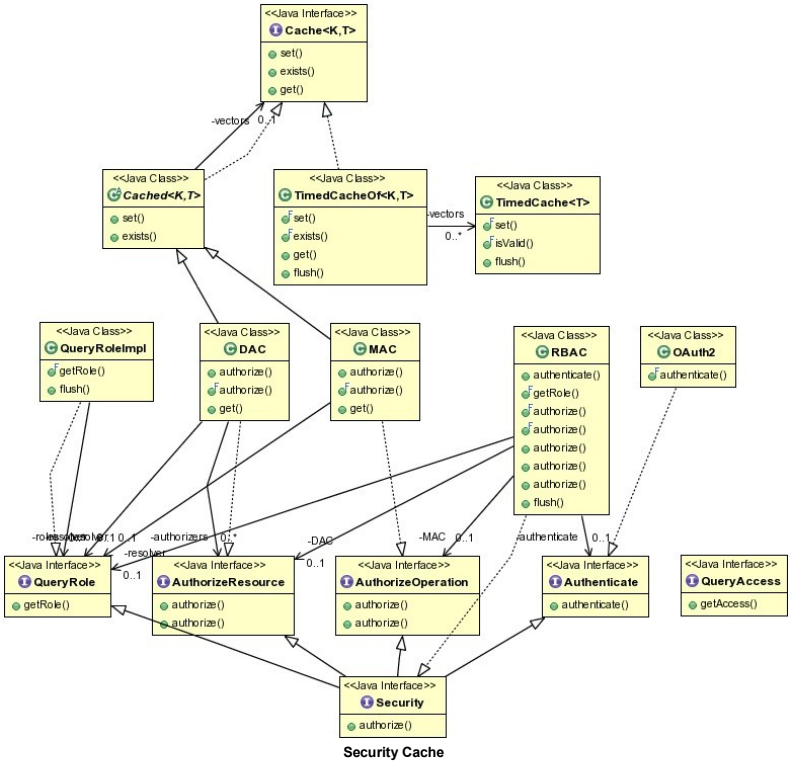
Security Interfaces

Security Module Diagram



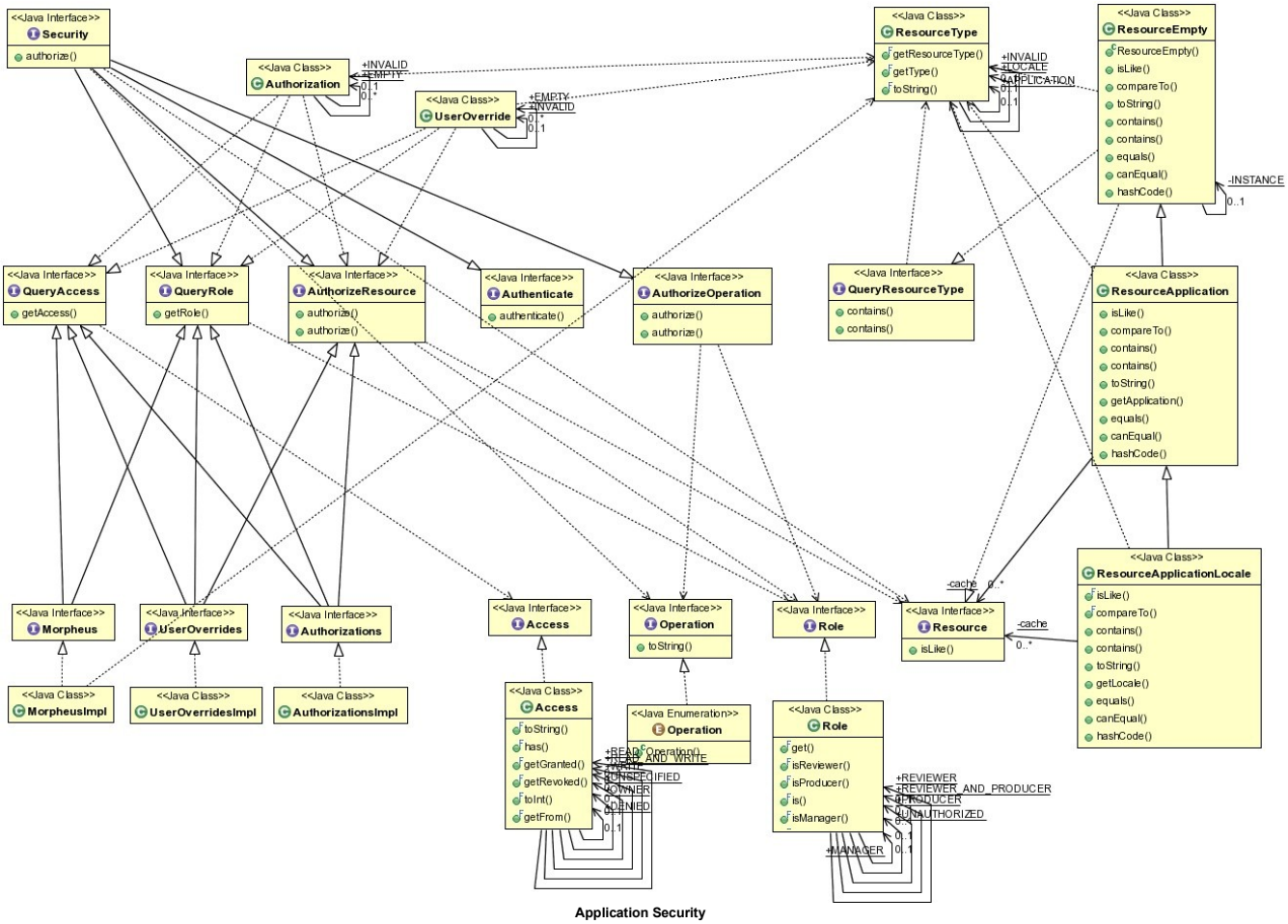
Security Module

Security Cache Diagram



Security Cache

Security Model and Application Implementer Diagram



Application Security

System Tests

The new AViK system was designed with many best practices and testability is actually a requirement in the L10N Tiger Team. There were some time constrains for the overall system development, but basically we currently have the following types of tests:

- Unit Tests:
  - If you think about it, the system really just moves data around, a lot of the data comes from the DataStore, and we also do input validation. Many of the classes for the entities we use simply contain data which we access through accessor methods. It makes no sense to write code to unit tests these types of object, not only would it be time consuming but it would serve no purpose, because we would in essence be testing the Java Virtual Machine.
  - So we have unit tests for the types of objects that most need it and our limited time allowed.
  - In the future we should write more of these tests. The other reason is that the system has been in a state of flux during development and it just would not be nice having to update the tests all the time
- Integration Tests:
  - Tests about 95% of the system locally, exceptions are: Morpheus, we use a mock object, but actually we could even test the real Morpheus, but it's not done currently. Web upload process, but will use mock system for this
  - We build the system as closely as possible as what would run in the cloud, we use a test feature available in JDO so that we exercise the app engine test features as well
  - Currently the tests perform CRUDL operations and some of the API calls are done as well
  - All security related code is also excersized
- System Load Tests:
  - These evaluate the performance characteristics of the system, excluding parts of the system, such as the environment in which it is to run
  - Allows us to observe the system more or less in isolation
  - The Java SDK has a tool that enables us to observe the internals of the system while in operation. We can see many things such as garbage collection, memory and cpu usage, where the system spends most of its time, thread details, etc...
- Multithreading Tests:
  - The system does not make use of threads directly. That is, we do not create threads for any type of purpose directly, though this might be done for us by the AppEngine system. This is beyond our control and can vary from AppEngine version to version
  - AppEngine allows us to configure our system as Thread Safe or not. This has a few performance implications. The best option is Thread safe. Thread unsafe would mean that for every API call if the system is busy servicing a previous API call, it would create a new instance of our application. This is extremely ugly and the cloud system would then shut them down again, and repeat this process. Even though the system is light weight, it doesn't mean we should ignore concurrency.
  - I've chosen to make the system Thread Safe, since it's the best option. To effect this I had to make a lot of the containers thread safe, by either choosing thread safe containers which are a bit less performant than their concurrency unaware counterparts, and by choosing immutable data whenever possible. I've also used the Iterable interface to pass around collections of objects, this also has the nice effect that most of the time objects are not copied unnecessarily.
  - A thread safe appengine app can be called for more than one service call concurrently. The appengine run-time decides on the algorithms and is beyond our control. This provides for a better user experience since the in-memory cache system can be reused. Also, the system does not have to be stopped and started too frequently as load increases.
  - I found that there were some limitations in these types of tests due to the way the JDO library works and the implications of how threads work when accessing data.
  - The JDO DataStore testing library documents that it is unable to handle concurrency, though it is thread save when deployed and when dealing with a 'real' DataStore. So basically, I understand this to mean that its mock objects are not thread safe
  - I implemented a small set of mock objects so as to replace the JDO library so we can test concurrency. We're able to do this relatively easy because of the modular design
  - However, to my surprise, it turned out that for some reason the concurrency safe containers weren't as safe as I thought. So currently we're able to test just about everything concurrently, as long as we can somehow add better synchronization at the mock object leve. This is not implemented yet, and would not be any time soon
  - What the system can test concurrently is CRUDL operations where the data is expected to be found
  - I'm thinking that this actually may be working correctly. Let me list a use case scenario for one concurrency test: the code runs an insert and then a delete, sometimes the operations fail. This actually makes sense, the failure is NOT a system failure but a failure to find the record because it was simply not there at the moment. Why? Thread 1 inserts record A, thread 2 inserts the same record A, thread 1 deletes record A, thread 2 attempts to delete the record, and guess what? it's no longer there. So the errors I see are the ones I generate and log. However, the logging mechanism is slower, much slower than the system
  - What the multithreading test illustrates is that we must be very careful when dealing with the actual data, and this is a limitation that all NoSQL databases share, the problem with what they term 'eventual consistency.' Nevertheless, this is an issue that we are not very likely to find, specially in the initial stages, because a) we don't have that much data that changes from under us, you can think of the system as a kind of blogger where the reviewer adds information to it, nothing really should ever be deleted except for admin purposes, and b) there is no b, that's basically it, actually caching helps somewhat to alleviate the possibility of data collisions at the expense of consistency, but as we said before NoSQL databases, which the Google data-store is one of, apparently currently does not guarantee consistency, but again no issue here, since we said we just add data

Test Results and System Performance

Just some preliminary test results and performance numbers

- Please take notice that these tests exercise the system as a whole, including the security module
- The system is very highly performant. However, currently the test data set used is quite small I'm guessing probably about 0.1% of what we will be dealing with for day to day operation.
- Load Test with list operation for applications with a data set of 6 records, using the mock system. About 1,000,000 operations per second.
- Same as above but using the actual JDO library with its test module. About 100,000 operations per second
- Multithreading test with mock objects, firing over 1,000 threads is also very fast, performing list, insert, delete, retrieve. When operations fail because data is not there it slows down the system significantly because packaging the error message is more expensive than the rest of the run time. The threads run loops of about 10,000 iterations each and it goes anywhere from 40 to 120 seconds
- The same test using JDO test library is much more slower because it not being thread safe causes exceptions which are very slow comparing to anything else. But we do catch all exceptions and process the stack trace and log it. So for this test I had to fire a lot less threads and smaller loops.

Note on System Design

When I was developing the system, I had other systems that we thought we would rewrite, so for this purpose I made the system more modular because I thought that it may also be useful for other applications I might rewrite. Because we have a tight deadline I had to focus more on this system and the other systems were assigned to other team members. The other system is VAR and it currently suffers from low performance and other issues. So the system may seem a bit overkill for such a simple app, but I wanted to explain the the initial purpose was to design it so that it could be reused. This may still be the case and it could be reused as a generic application framework for the Google technology stack for l10n team, or just the security module.

System Design

The following lists other details and contains links to Doxygen automatically generated documentation. Following the links, you would see some diagrams for most. The diagrams are two kinds:

- Class Relationship Diagrams, automatically generated, and
- Sequence Diagrams

Main sections of code:

- The entry point to the system is the Google Endpoint
- Everything centers around the Controller
- The Strategy (API Commands, instance) pattern is used to dispatch the command and/or event.
- The Mediator pattern is used to converge at a point where the operation is ready to proceed without distractions or delays.
- The Relay , is something I sort of made up, and it looks like a pattern but, there may be a better way or more elegant way to handle that, but I had to deal with Java's apparent limitation. If I had implemented this as polymorphic types the boiler plate needed for all actors would have been, hilarious, so no thanks Java, I do my own thing, thank you.
- The concept of Actors is not to be confused with the functional programming concept of an actor, here is simply a storage handler for a particular entity type, for example the Actor Applications handles the entity Application , Reviews handles Review, and so on.
- Wrappers are used for objects Parameter and Product because they will adjust for the relevant type as necessary, and in that sense they are polymorphic, but not really using the Java polymorphic mechanism of inheritance, but of generics.
- The Composite object, in our application is a group of records/entities related to a particular app, such as all screens and locales, data only not the actual images. This allows us to provide an API that is less 'chatty' and more performant.
- The Result class is used to return concise information to the user in a consistent way
- The Null Object Pattern is used in most cases as appropriate. This allows the code to be simpler in such a way that we don't worry about nulls. In the event of an error, the user will receive an empty object and this would indicate the error condition. And later we might also return a message if appropriate, but remember that this is mostly for CRUD operations which are mostly administrative in nature and don't deal with the end user directly.
- Caching is done in 3 ways, two in the back end, the one deals with caching in the Storage handler, and the second is more useful and tailored to our particular need, where we cache a composite object.
- Initially, I had used an input validator from the Hibernate/Validator sub-project. However, in my system load test, I observed that it was using about 68% of CPU time, so, no thanks hibernate, we'll roll our own.

Operations and Security Related classes

- List of all Operations and their Operation Attributes
- List of Access types
- All Roles allowed in the system
- The hierarchy of all Resource types. ResourceEmpty , ResourceApplication , ResourceApplicationLocale
- The Authorizations DataStore persistence handler and the Authorization entity from model
- The UserOverrides DataStore persistence handler and the UserOverride entity from model
- RBAC
- OAuth2
- MAC
- DAC
- RoleEnquirer

Current Code Base

SLOC/Directory

- 1555 bll
- 1207 model
- 911 security
- 825 storage
- 753 test
- 593 setup
- 403 collaboration
- 277 api
- 159 util
- 139 services
- 105 cache
- 16 top_dir
- 502 security module

Totals:

- java: **7445**