**CS673S16 Software Engineering** 

**Team 1 - Project Name**

**Software Design Document**

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member | Role(s) | Signature | Date |
| Steve Black | Team Leader, Design Lead | *Steve Black* | 12/12/2016 |
| Anthony Valencia | Configuration Lead | *Anthony Valencia* | 12/12/2016 |
| Demetrius Glover | Environment and Integration Lead | *Demetrius Glover* | 12/12/2016 |
| Yash Bavishi | Requirement Lead | *Yash Bavishi* | 12/12/2016 |
| Kevin Ding | Implementation Leader | *Kevin Ding* | 12/12/2016 |
| Kevin Sperling | QA Lead | *Kevin Sperling* | 12/12/2016 |
| Bakyt Bakayeva | Security Lead | *Bakyt Bakayeva* | 12/12/2016 |
|  |  |  |  |

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Change** |
| 1 | Steve Black | 10/24/2016 | Iteration 1 |
| 2 | Steve Black | 11/14/2016 | Iteration 2 |
| 3 | Steve Black | 12/12/2016 | Iteration 3 |

[Introduction](#_87t9hln2vjz0)

[Software Architecture](#_buttcq9i221r)

[Design Patterns](#_x18fj36s1121)

[Key Algorithms](#_mtfbusfb0eq3)

[Classes and Methods](#_7ucksmkf6rzx)

[References](#_15tmymhipvdv)

[Glossary](#_8n34lvocupub)

# Introduction

This document will outline the general design that we have applied to our project, which is to add open API functionality to the existing issue tracker software, to use that API to increase the integration between sub-products (requirements tracker and/or chat client) and allow for external integrations, to simplify the deployment process into a modular system that can be rapidly deployed in a Docker container, and to replace the chat client with a better integrated backend. Our goals in selecting this project were to increase the value of the suite of applications by improving the integration between the individual subproducts, and to allow the software suite to become parts of larger collections of software by opening the API to allow developers to pull information from the issue tracker in a simple, real-time manner.

# Software Architecture

The existing issue tracker software utilizes the Django Python web framework to present a web interface over a database for tracking issues in software (or other) projects. This was a shrewd decision made in order to enable subsequent groups of developers like ourselves to extend the project. As such, the choice has paved the way for what should be a smooth development path for our team. In addition to the issue tracking software, there are also a related communication tool (chat client) and a requirement tracking tool as part of the same overall suite of applications that deploy together that are also implemented using Django. For details about the original implementation, please see Team\_Documentation/Team\_3/project\_documents/CS673\_S15\_P13\_SDD.docx in our source repository. Details about our extension of this project will follow in this document.

Our project primarily entails making changes to the issue tracking software (in the early iterations) and also replacing the chat software with a Django-based implementation and integrating the two pieces more tightly. Generally speaking, we will modify the URL routing of the web application to support several new groups of API endpoints, and add several new views over the data model to support the new data collections to be returned in calls to the API.

Django REST framework module allows developers to define custom interfaces over data collections by extending the ModelViewSet class, while keeping the fields between the JSON representation and the database model the same (it is a very thin layer). For the GET methods of our API, we will extend the ReadOnlyModelViewSet and apply filter\_backends (specifically, SearchFilter) and search\_fields (to map query parameter keys to data model property names). The view set for obtaining a list of Issues that match our search criteria will be IssueViewSetRO, and the one for obtaining a single issue by ID will be IssueViewSetSingle. The former will be registered to URL routes of the pattern rest/issue?<param>=<value>&<param>=<value>; the latter will be restistered to /rest/issue/<issue number> for simplicity. More details can be found in the Design Patterns section of this document.

To accompany the opening-up of the API, we will increase security of the application by implementing token-based authorization, by which the caller requests a token with a username and password of an actual user in the database, then uses the token as authentication for subsequent calls. More details can be found in the Design Patterns section of this document.

Initially, we elected to “Dockerize” the application to speed the development and testing process. However, we made the concept of putting the software into an easily-deployable container a core goal of our project, so that future developers AND users can get the entire suite of applications up-and-running swiftly and without significant investment of operations time and resources.

Lastly, we will improve and simplify the architecture of the software suite by replacing the existing Node-based chat client with a Django-based implementation that will more seamlessly integrate to the other features of the software bundle. Just like the move to put the software into a container, this decision will greatly simplify the dependencies and deployment of the application.

# Design Patterns

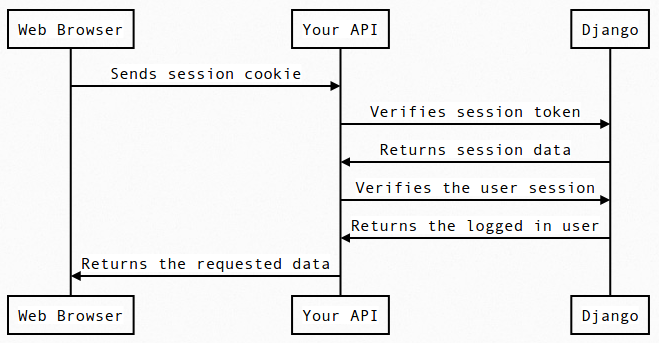
Our primary goal is to create a RESTful API that can allow requesters to access information from the issue tracking system and receive it in JSON format, providing all necessary data about issues in an easily-consumable format. To accomplish this, we paid special attention to organizing our resources into logical categories so that the API is intuitive and so that the CRUD functions (create, retrieve, update, delete) are predictable.

Our design will incorporate several different HTTP request methods that map to these CRUD function, including POST (for new issues), GET (for existing issues), and PATCH (for updates), using URL parameters and payloads as necessary when details are required in the request. Since some of our goals include reducing the result set of issues to those that match certain criteria provided by the requestor, there will be a filtering element in the design.

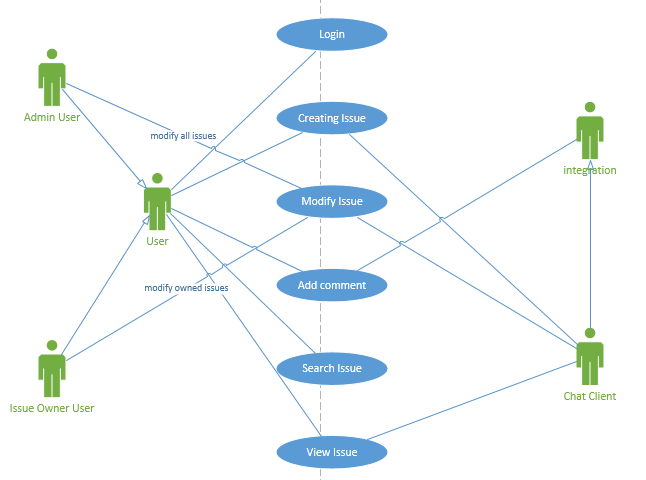
Returning a proper set of useful error codes and messages will also be an important part of the design, as it will make development intuitive for developers.

From a security standpoint, as we proceed through our development iterations, we will introduce security measures such as API authentication. As a best practice, we will ensure that state plays no role in authentication, and require HTTP Basic Authentication for any requests, at least for POST calls (or PUT or DELETE, if we decide to implement those).

To accomplish our goal of developing a useful interface for external developers, we will also create clear documentation for formatting requests and understanding the general structure of the API.



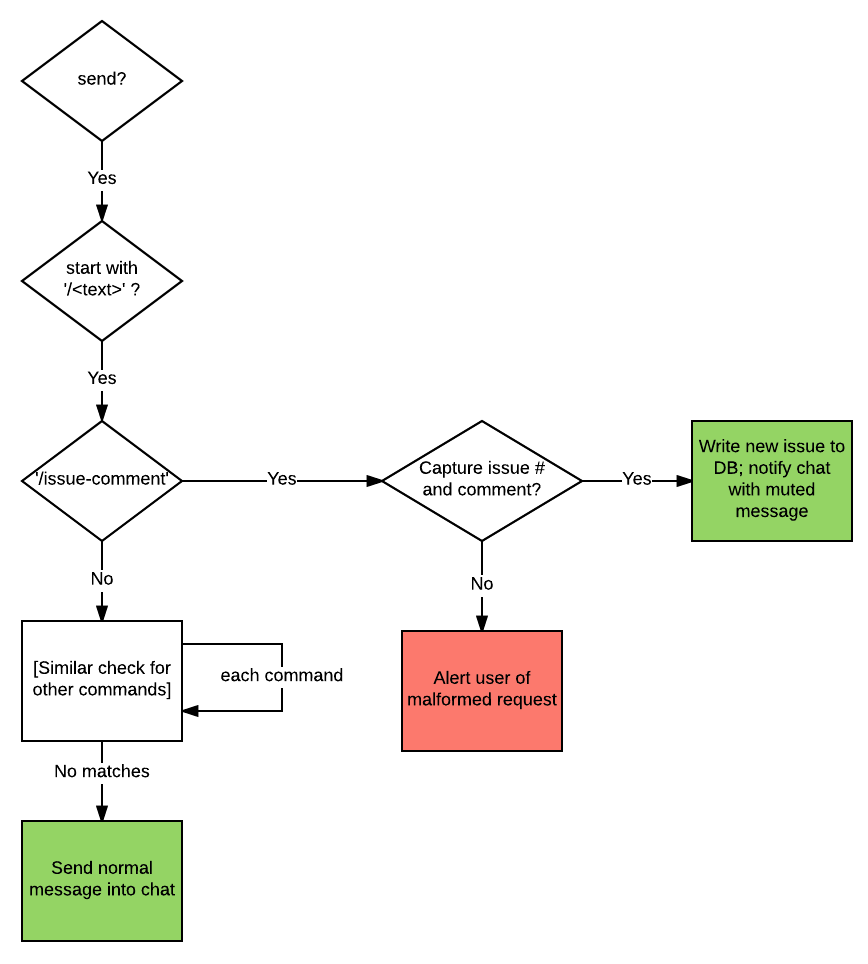
The design pattern for putting the software into a Docker container is fairly straightforward: the existing dependencies were already clearly enumerated, so they were encapsulated in a programmatic deployment step. From there, it is possible to issue Docker commands to deploy the build and start the server all in a simple step, and to do things like populate the database with demo data, if necessary (the latter is made possible by elements of the Django framework).



Use Case Diagram

# Key Algorithms

The nature of the requirements did not necessitate the develop of any complex algorithms. However, the adaptation of the chat client to handle specialized commands in addition to traditional plaintext messages required some regular expression analysis with matching and capturing of important commands and their arguments. The algorithm for managing this feature in the chat client is as follows:



# Classes and Methods

Our project does not intend to add any data models to the existing project, since the spirit of our goal is to increase *access* to the data, not to add to it. However, some important classes for defining the composition of the objects that are returned to requestors are as follows:

class IssueViewSetRO(viewsets.ReadOnlyModelViewSet):

"""

Obtain a list of issues that meet a certain set of search criteria

"""

queryset = it\_models.Issue.objects.all()

serializer\_class = serializers.IssueSerializer

filter\_backends = (filters.SearchFilter,)

search\_fields = (

'id',

'title',

'issue\_type',

'status',

'priority',

'project',

'reporter',

'assignee',

'verifier'

)

class IssueViewSetSingle(viewsets.ReadOnlyModelViewSet):

"""

Obtain a single issue that matches the issue ID provided

"""

queryset = it\_models.Issue.objects.all()

serializer\_class = serializers.IssueSerializer

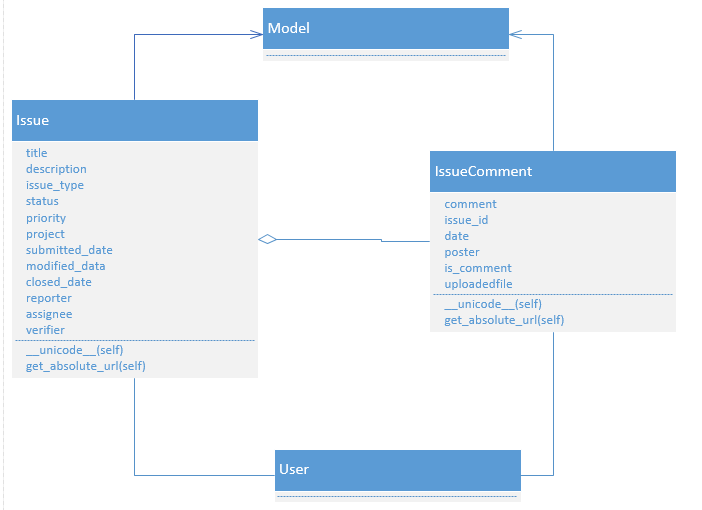
def retrieve(self, request, pk=None):

"""Return a single Issue"""

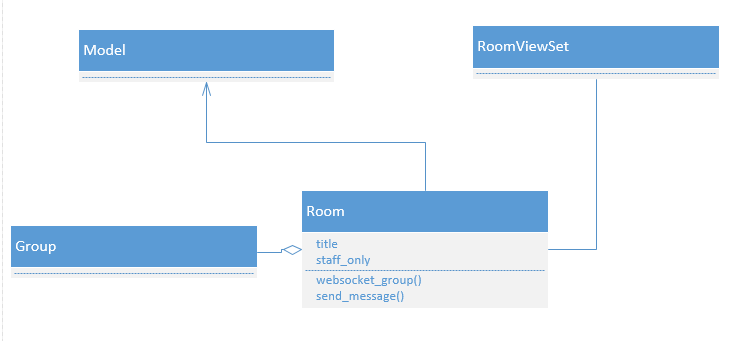
user = get\_object\_or\_404(queryset, pk=pk)

return Response(serializer.data)

Issue tracker model class is the main class to manage two basic classes of issue tracker. It sub classed into the Issue and IssueComment models, used to represent the data of the application’s main component issues and their respective comments. The default Django user model was used for the project. Noticed that “uploadedfile” is a new attribute added to the IssueComment class for previous model, which represent the file user uploaded for each comment.



Chat model class has been totally changed. Comm (node based chat application) is removed, chat (django channels based chat application) is added to the model.



# References

Best Practices for Designing a Pragmatic RESTful API: <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

Docker: <https://docs.docker.com/>

Django Viewsets: <http://www.django-rest-framework.org/api-guide/viewsets/>

# Glossary

* + API: Application Programming Interface
  + REST: Representational State Transfer
  + Docker: Development and deployment system that wraps software in self-sufficient file systems, allowing for simple, platform-independent deployment