**CS673S15 Software Engineering**

**Group Project – Requirement Tracker**

**Software Design Document Team Alpha**

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member | Role(s) | Signature | Date |
| Max Almanzor | Design Lead | *Max Almanzor* | 4/23/2015 |
| Vikrant Sanghvi | Design Lead | *Vikrant Sanghvi* | 4/25/2016 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Change** |
| 2016 v1 | Vikrant Sanghvi | 4/25/2016 | All diagrams and various section with the new team design improvement and functionality. |
|  |  |  |  |

[Introduction](#h.87t9hln2vjz0)

[Software Architecture](#h.buttcq9i221r)

[Design Patterns](#h.x18fj36s1121)

[Key Algorithms](#h.mtfbusfb0eq3)

[Classes and Methods](#h.7ucksmkf6rzx)

[References](#h.15tmymhipvdv)

# Introduction

This document describes the architecture of the requirements management system including the major components and how they interact. The design goals for the project were:

1) Create a system that fulfills the requirements for a requirement management systems using Django as our framework. Django was chosen as the platform for our group so using it is another requirement for our system.

2) Use an architecture that would allow developers unfamiliar with our framework to contribute. Our project uses the django framework, this decision was made with consensus from the three teams in our group as the best choice for our project. However, no one on our team had used django before and only a few of us had used python. Therefore it is import that

our design use a simple approach that we could pick up quickly instead of an elegant solution that would be more difficult for the group to understand. This design choice means that our project has a few places with code that could have been written using design techniques like polymorphism or used more advanced features of the framework but were instead done simply as a means of quickly writing code. If the project is carried forward then this could mean there will be some necessary refactoring to make the code more maintainable.

3) Allow easy integration with other teams. The other two teams in our group take advantage of the user management and project management features we developed. We needed these

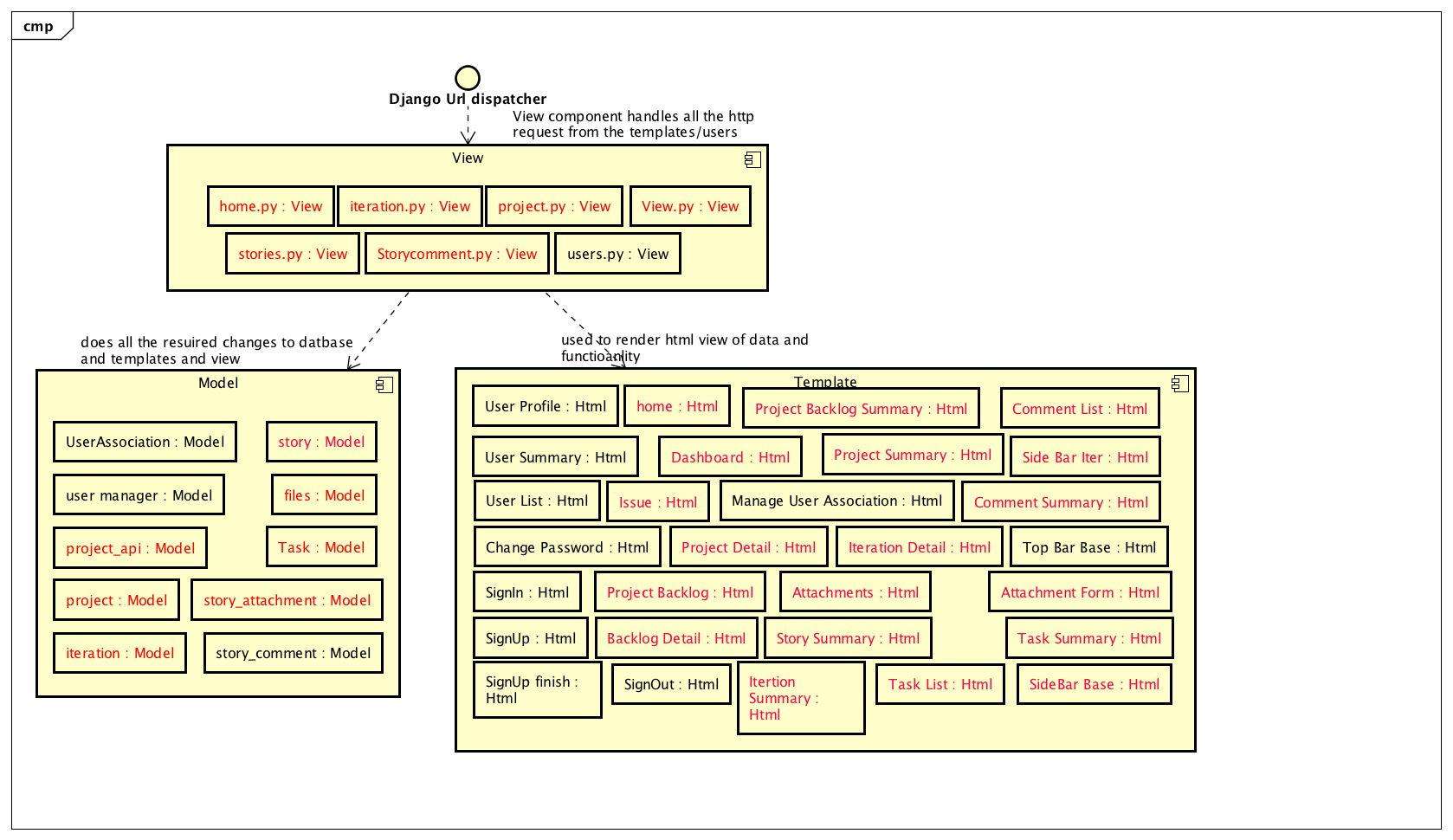
components to be easy to use and extend without knowing how the other teams planned on implementing their components. This meant that we had to add easy to use api methods that could be invoked from outside of our team's application and well as used internally.

4) Favor speed of development over performance. From the beginning we knew that our system would not need to support many hundreds of concurrent users. A work load like that should not be too difficult for a system using our technology stack to support. Therefore we will not make architectural decisions based on performance and instead focus on ease of implementation. This goal also frees up team members to spend their time adding features instead of doing performance tests or profiling code.

# Software Architecture

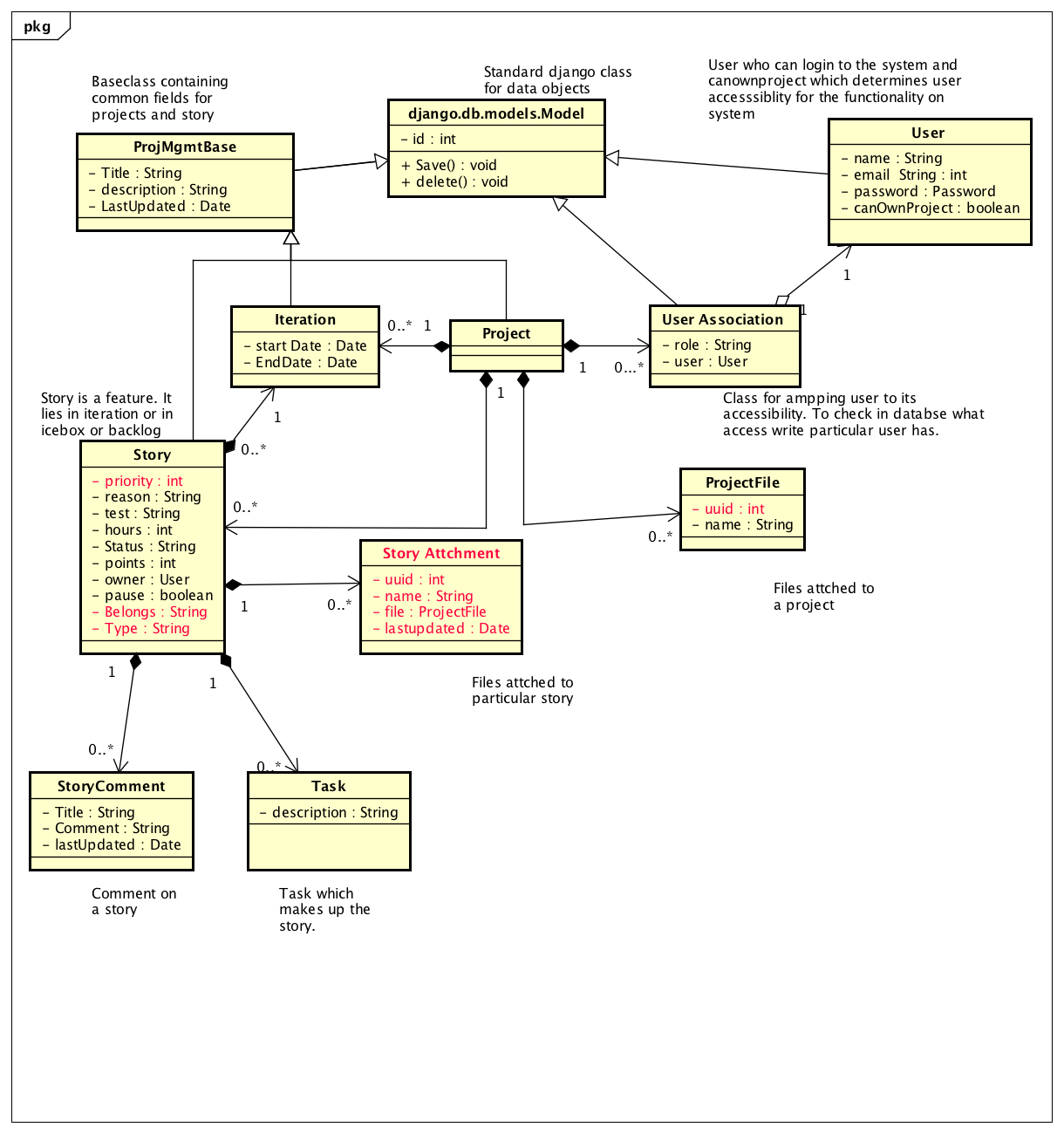
The requirements management system is build on top of the django framework. Django is a python based framework for web applications that provides convenient abstractions for url processing, html rendering, object relational mapping and other features. Our architecture follows the django model, view template pattern, which is very similar to the well known MVC design. In this architecture paradigm the code for creating, updating and persisting entities is done in the model component, in the exact same way as MVC. Entity classes are defined extending django.db.models.Model allowing automated schema generation and persistence, similar to the Java Persistence API. The view component is responsible for reading data from http requests, reading and writing data using the model and deciding which template to use to display data for the user. In django the view is similar to the controller in MVC. The templates are html documents with snippets of python mixed in. The python portions execute on the server side to convert data passed in from the view into renderable html. Templates in django are comparable to the view in MVC.

The following component diagram shows a general overview of the major modules in our system and how they interact.



Model - The model component contains modules for entities and apis for accessing those entities.

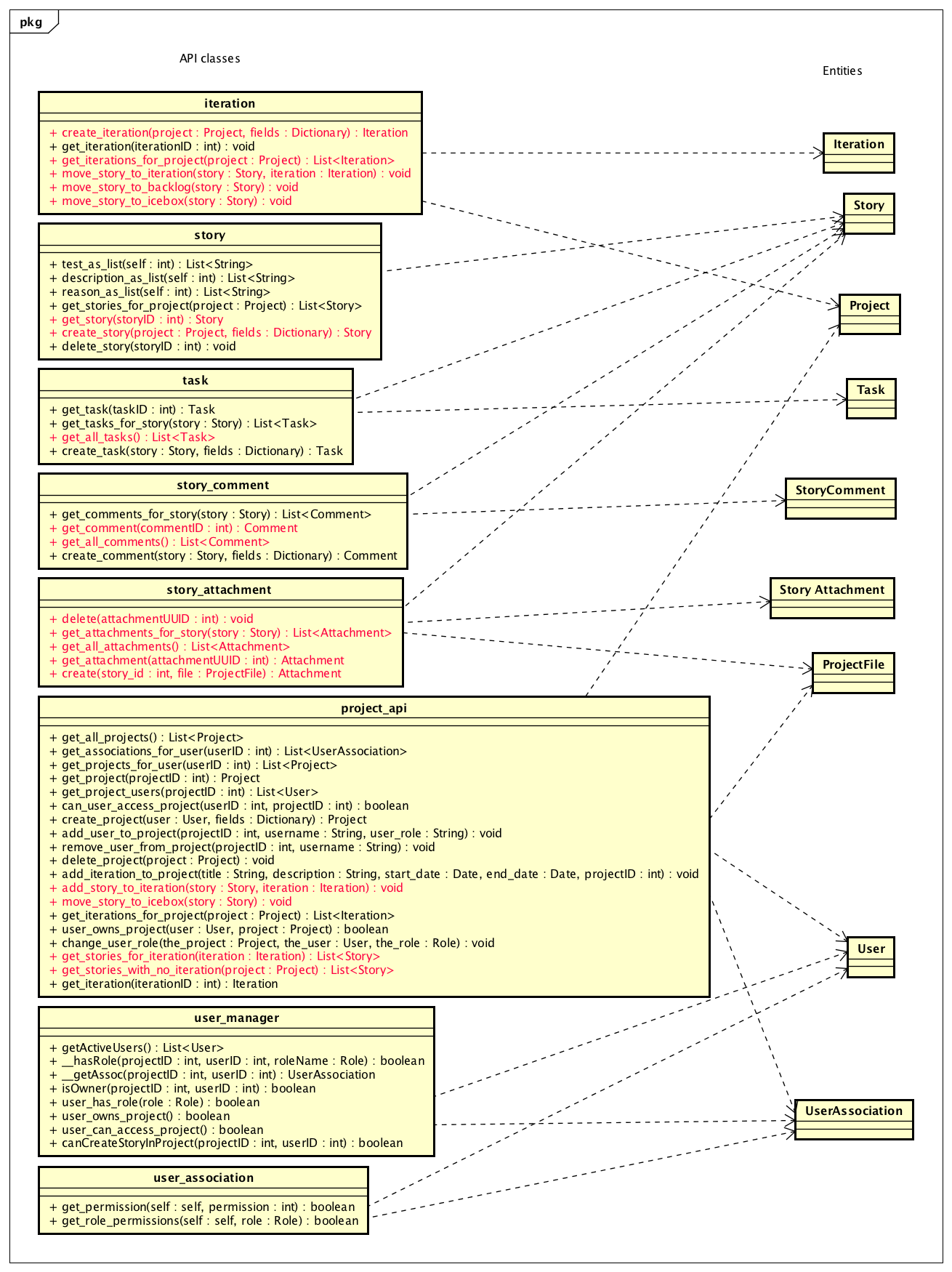
Here are the entities in the requirements management system:



* django.db.models.Model - Base class provided by django that provides object relational mapping.
* ProjMgmgtBase - Base class for our system that provides a few standard fields that will be required across several classes in the requirements management project.
* Story - A user story. Contains hours, priority, tests as well as other relevant fields.
* StoryComment - A comment added by a user to a story.
* Task - A task added to a story detailing a small piece of work.
* Project - A collaborative piece of work involving multiples users and features. A project has stories, associated users, attachments and iterations.
* User - an individual that can log into the system. This class is provided by django and is used by other classes in our system.
* UserAssociation - class used to implement a many to many mapping between users and projects. A UserAssociation contains a role that determines what permissions a user has on a project. This supports the requirement that a user can have different permissions on different projects.
* ProjectFile - A file attached to a project.

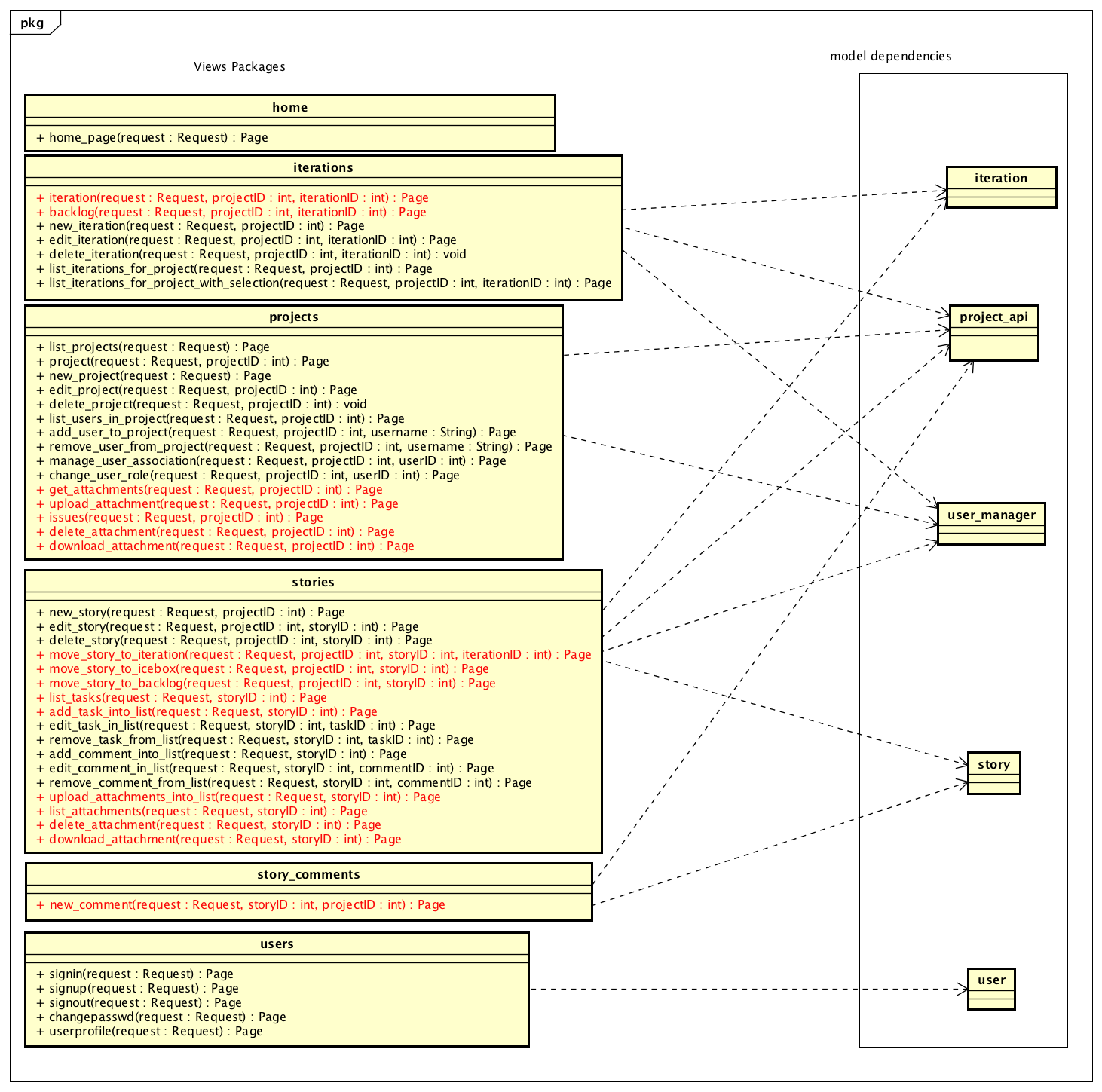
Each entity class is stored as a table in the database created and maintained by the django model ORM layer.

Entities are exposed to outside components through api python modules which contain methods for creating and working with entities. The following diagram shows the api modules as well as the entity classes they depend on.

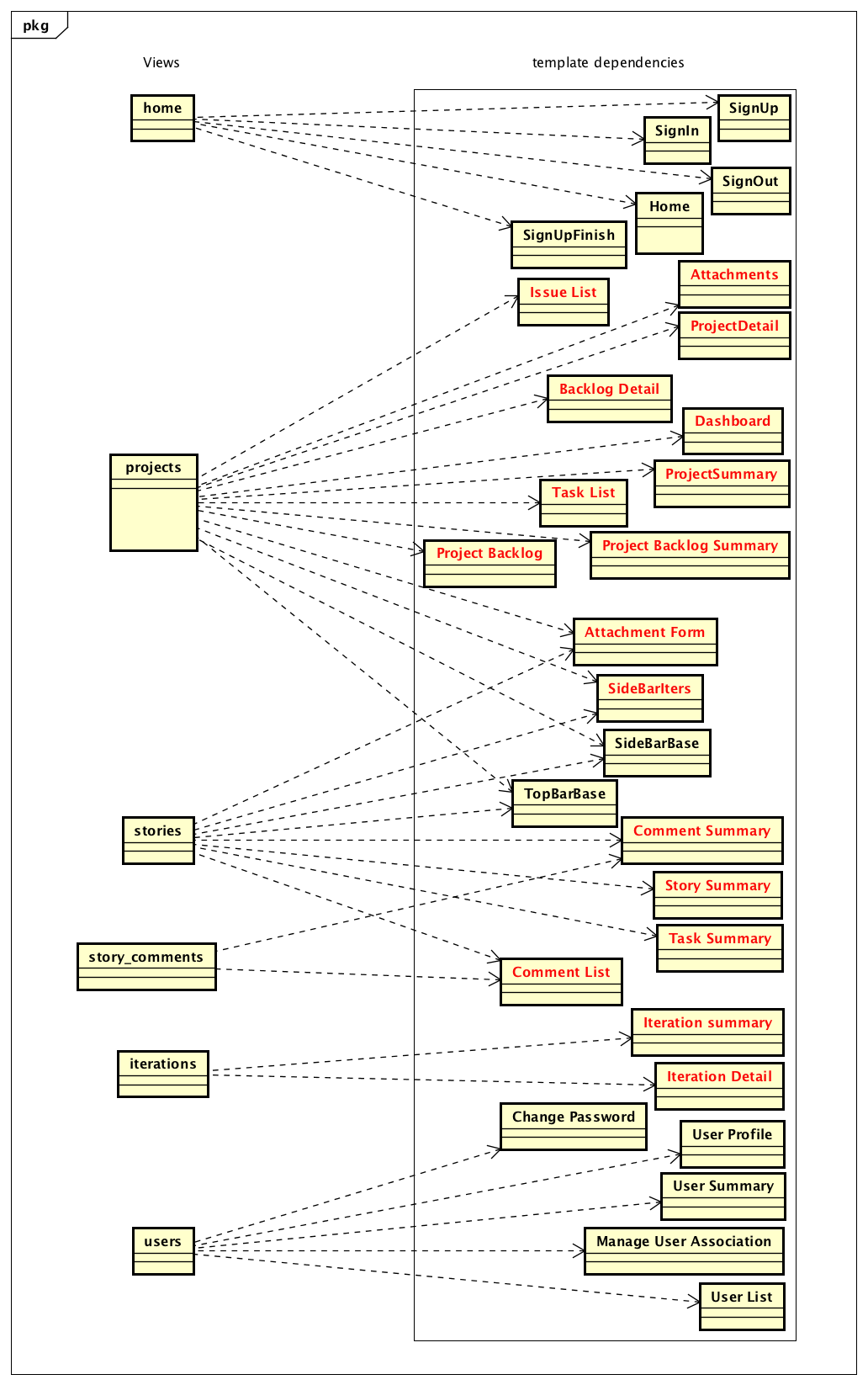


View - The view component contains the functionality for processing http requests, working with the model layer to read/write data and using the templates to render html. Views are functions in django module that taken in a http request and optional http url parameters and return http responses.

The following diagrams shows our view modules as well as the model api classes the views depend on.



Template - Templates are used by the views to render html responses. The following diagram shows the view classes and the templates they depend on.



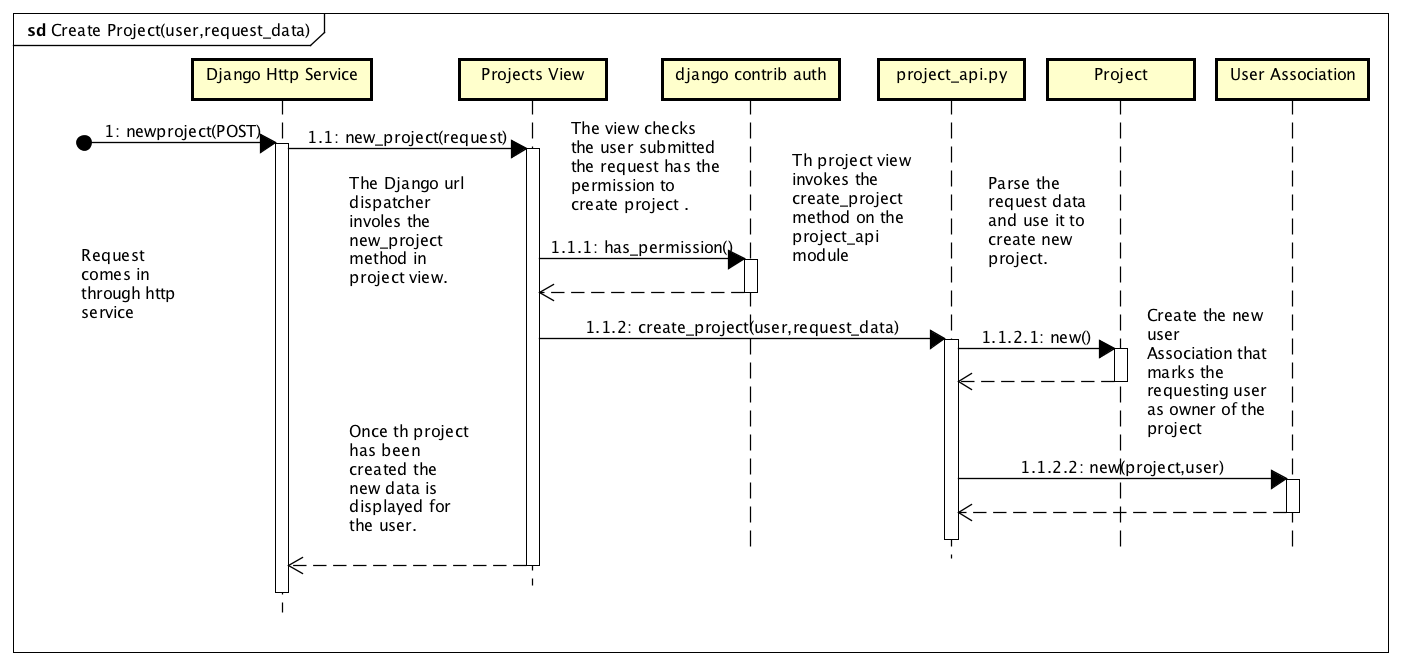
# Design Patterns

The primary design pattern that we used is **the Model View Template** pattern that I described in the Software Architecture section. We used this pattern to drive the overall structure of our program.

We also made some use of the factory pattern to create entities. The api classes described in the software architecture section expose create methods used to generate entities given the passed in parameters. This pattern allows our project to have a single location where an entity of a certain type can be generated. It also gives us flexibility for future extensibility. If the requirements of our system change and we need to change implementing class for a certain entity we can modify the factory method to return a different type that maintains the behavior of the original class. This would give us the desired behavior change without requiring changes to other classes in the system.

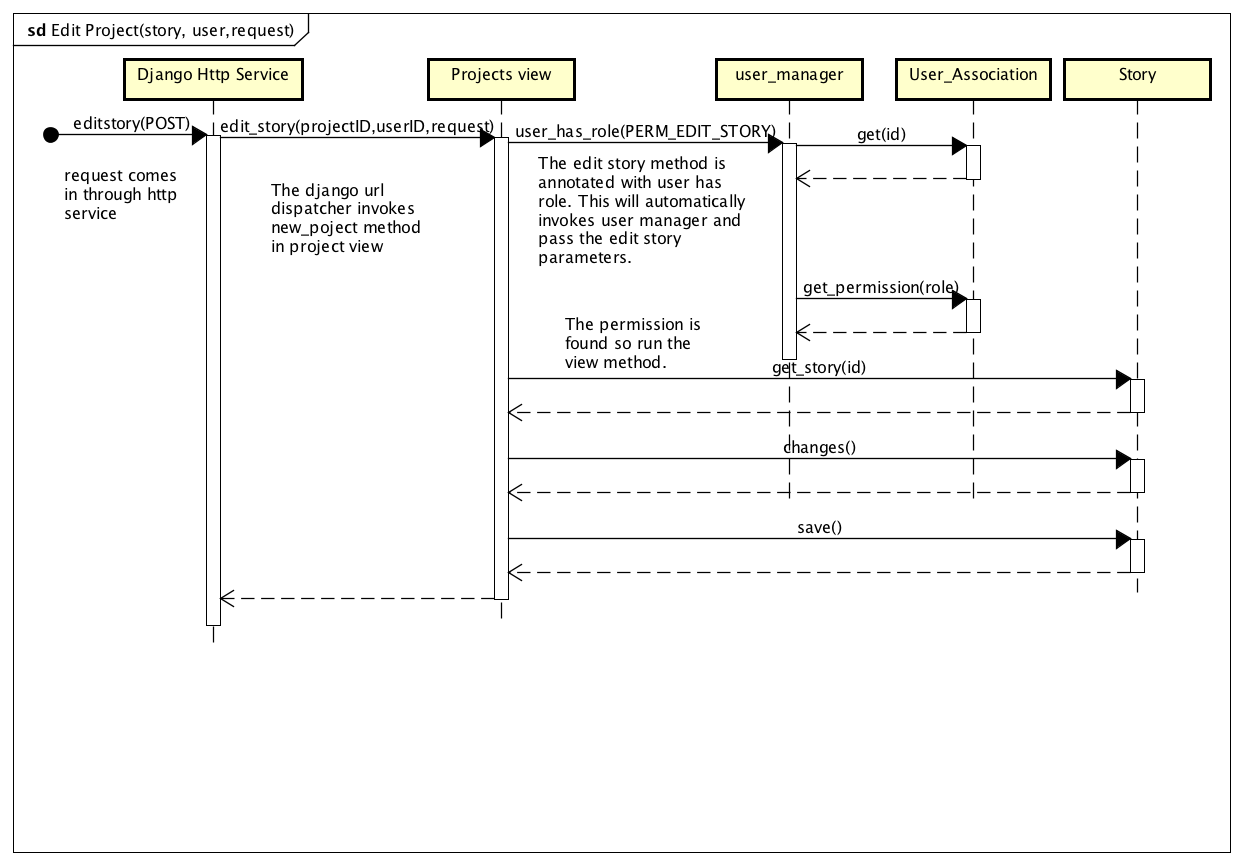
# Key Algorithms

The key algorithms in our project are based around permissions. When creating a project we need to make sure that the user has the permission to create a project.



The create project request comes in through the django http service and is routed to the correct view method on the projects view. The view method is annotated with a decorator function that checks the user associated with the request to see if the user has the permission to create projects. If the user has the permission then the view uses the projects\_api class to create a new Project and a UserAssociation listing the user as the owner of the project.

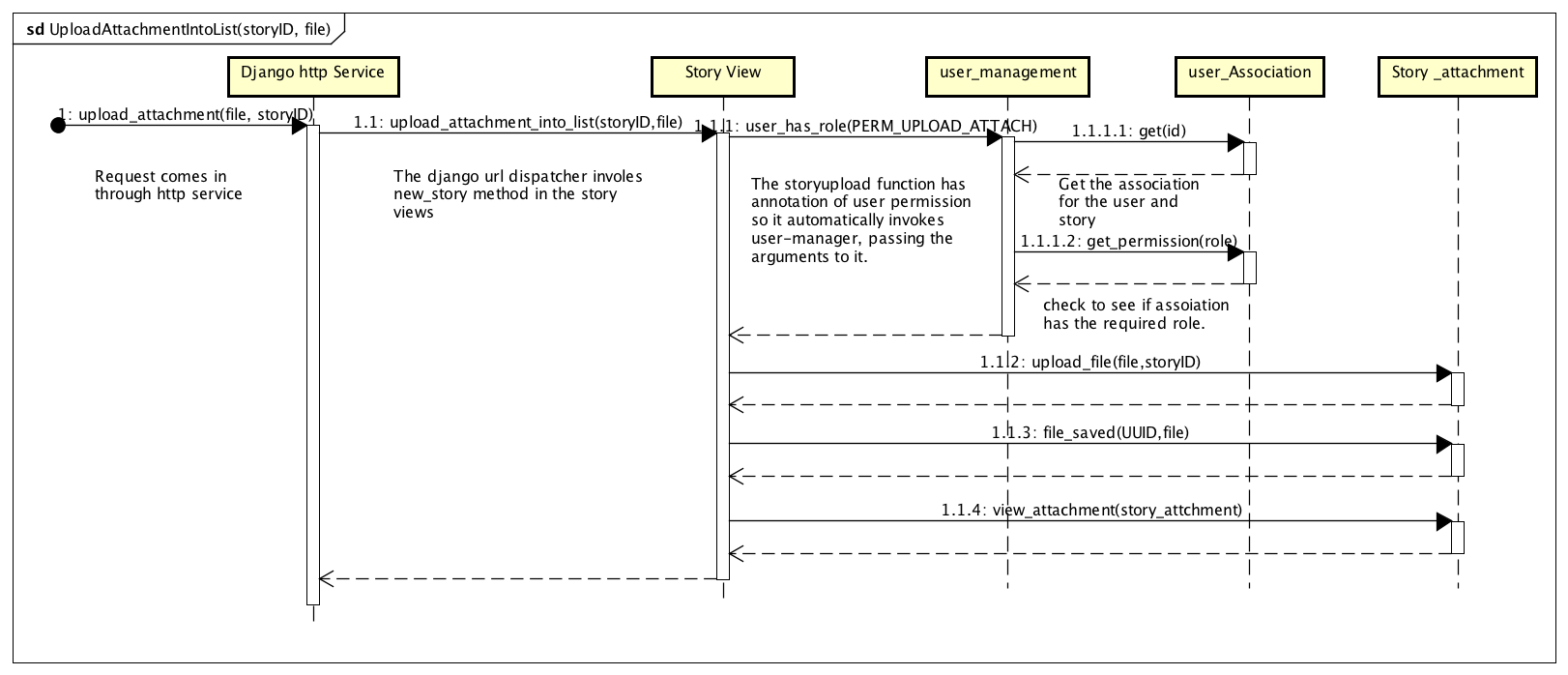
When editing a story we need to check the UserAssociation linking the user and project and make sure it contains the correct role.



The edit story request comes in through the django http service and is routed to the projects view. The view method is annotated with a decorator function that invokes the user manager which finds the UserAssociation object and checks the contained role. If the role gives the user permission to edit the story then the view method proceeds and story is edited. If the user does not have the correct role then an error is thrown. A similar process is followed for other processes such as creating stories or pausing them.

One of the other algorithm used is upload file or attachment to story. Instead of just uploading file to project, we decided to attach file to user story to make it more meaningful and easy to interpret and organize. For uploading file to directory and maintain a database for it, we choose to generate uuid identifier for files and store them in database giving more efficiency in running. Also uuid generator is supported by Django framework which made the work easy.

We need to check the user association even more adding attachment to story.



# References

Factory Pattern - <http://en.wikipedia.org/wiki/Factory_method_pattern>

Django Documentation - <https://docs.djangoproject.com/en/1.8/>

Model-View-Controller –

<http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

* All diagram have red labels to mark the changes done by our team in the existing project design. And diagrams are built using astah online tool.

# 