Studentable

*A University Course Registration Management System*

Early Design Document

*ETPB Software Design™ | 212 McAlister Dr, Saint John NB*

Table of Contents

[Preface 3](#_Toc57995057)

[Introduction 3](#_Toc57995058)

[Software Design method 4](#_Toc57995059)

[Design Considerations 5](#_Toc57995060)

[Abstraction 5](#_Toc57995061)

[Modularity 5](#_Toc57995062)

[Information Hiding 5](#_Toc57995063)

[Complexity 5](#_Toc57995064)

[System Structure 6](#_Toc57995065)

[Components 7](#_Toc57995066)

[User Interface Tier 7](#_Toc57995067)

[Logic Tier 8](#_Toc57995068)

[Database Tier 9](#_Toc57995069)

# Preface

Within this document is the outline for an early version of a university course registration management system. This document will focus on the design aspects of this software system, primarily the technologies used and the decisions behind using those technologies. As this system evolves, so may this document. The intentions of *ETPB Software Design™* with this projectare to provide an example of how a software system should be built and what steps should be followed. It is purely educational and will in no way be used in a professional setting. All source code for the project will be in the Github repository that also hosts this document.

# Introduction

Studentable is a web-based course registration management system. Its goal is to provide a user-friendly platform for students, professors and institutions to keep track of and organize their many courses and the students within them. The decisions made for the design of this system, which will be further explained later in the document, were all made to support an agile and simple product design. There is no need to overengineer something that students and institutions will rely on, therefore we opted for a method of development which was agile and allowed for continual improvement, but also validated our system at each step to make sure we were building the right system.

# Software Design method

The development of Studentable follows closely to the Extreme Programming (XP) approach. This is a pure agile method which focuses around a few fundamental best practices. For our purposes, we followed the following best practices when designing and developing out system:

* Small Releases:
  + We intent to release Studentable at first as a small system with its *essential* features (student/professor authentication, dashboard, database integration). Once this is accomplished, further versions will be released in short cycles which addon to the original release.
* Testing:
  + Testing will be a key factor when building this system. Due to the plethora of use cases a management system like this one brings forward; our team must ensure that we are always getting acceptable behavior from our software.
* Refactoring:
  + At first the system may be messy, but refactoring will be a routine task. This doesn’t necessarily add more to the software, but it does reduce complexity and improve quality which is of upmost importance.
* Continuous Integration:
  + The 3-Tier Architecture which was chosen for this project greatly supports the practice of continuous integration. When being developed, the system will be integrated and built many times a day to ensure we always have a working build.

Overall, the XP method may not be everyone’s first choice for a system like this. But with the unknown requirements that could possibly be brought forward by institutions who opt-into the system and the ease of communication with these institutions, the XP method will provide an approach that will allow us to quickly provide custom course management solutions to any University that hires us.

The goal with Studentable is to provide a base system, and then modify it for a given client based on their own custom requirements. This development method allows for this sort of flexibility.

# Design Considerations

This section details the considerations that were made while designing the components of our system to ensure simplicity, facilitate maintenance, and measure quality.

## Abstraction

The design of this system will employ a sort of procedural abstraction. Although it does not follow directly the main-program-with-subroutines architectural style, it does decompose its functionality into a set of defined tasks. First, we take in the user input from the graphical user interface, we authenticate the request, interpret the data, query the database, and return the request information to the user. This may be difficult at first since the programming language of choice for the back end of this application, Python, does not necessarily strongly reinforce abstraction. But since this is not the top priority, Python will still work fine.

## Modularity

Our system is divided into three primary components, the user-interface front end, the logic tier back end and then the database tier. The UI tier would have logical cohesion as this is a component that contains all the input routines and it doesn’t interact with itself. The logic tier would have sequential cohesion, where the input given from the UI is manipulated and interpreted through the elements of the logic tier before the result is achieved. Finally, the database tier would have communicational cohesion, each element of the database operates on the same external piece of data. We will strive for data coupling between each of these components, since only the requested information queried by the user needs to be passed in between each tier.

## Information Hiding

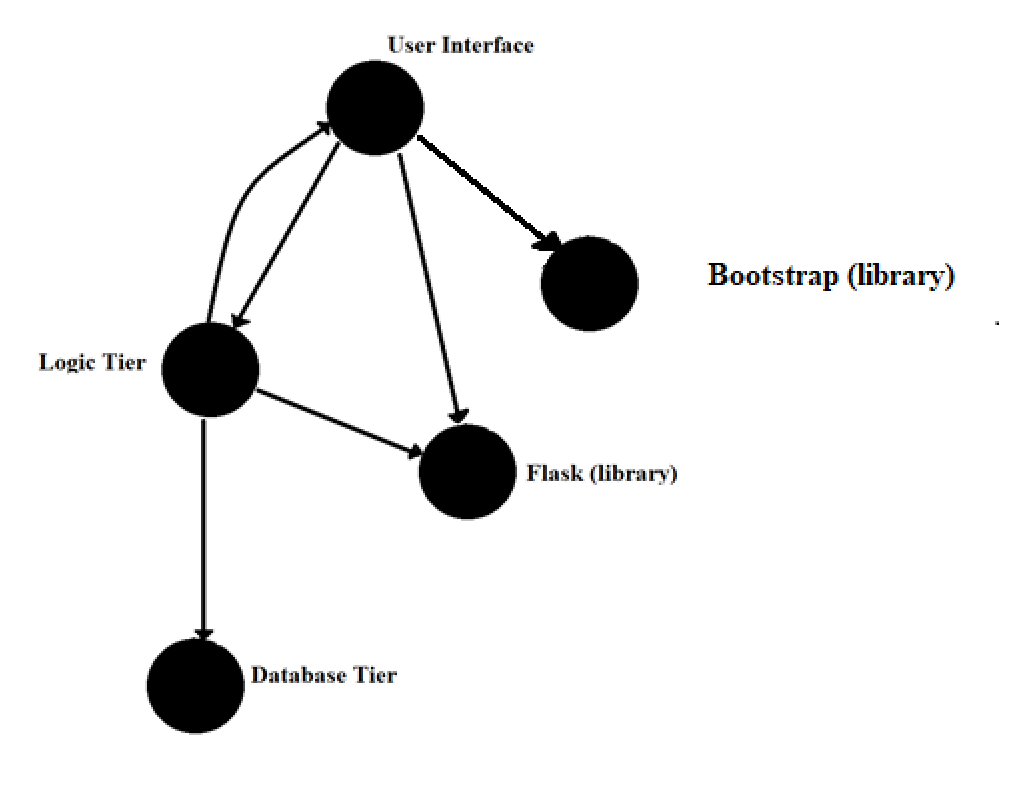
Ideally, we would like to hide as much information on the inner workings of the system as possible from the user on the UI tier. Their only concern should be the things on their screen and not how it is appearing. Therefore, we will employ strict restrictions on how users access this data (i.e. only through our dashboard). This is closely related to abstraction, which is true in our case as each component will hide some design decision from the component before it.

## Complexity

Our goal is to create as simple of a system as possible while still maintaining the functionality we are attempting to provide. This will require careful monitoring of both size-based and structure-based complexity metrics. We chose Python primarily because of how easy it was to write a web application and how few lines of code it really took thanks to the help of outside libraries like Flask. This will help keep the size of the system down. The 3-Tier architecture that was chosen to model this system will ensure that we maintain a reasonable control structure which will be easy to comprehend and maintain.

## System Structure

Here is an overview of the structure of our system and its dependencies in a graph.



* + Both the User Interface and the Logic tier use elements of the Flask library
  + The User Interface uses elements from the Bootstrap library
  + The User Interface invokes methods in the Logic tier and the logic tier returns data
  + The Database tier delivers data to the Logic tier

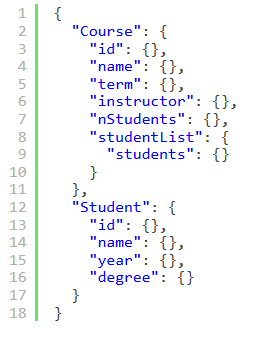
This graph has a tree impurity of m(G) = 1/3 = 0.3333333

# Components

This section details the three major components of the system and discusses their ten attributes as identified by the IEEE Standard 1016.

## User Interface Tier

* Identification: The User Interface Tier
* Type: A collection of html/css files
* Purpose: Provides a nice and clean user interface which allows users to interact with the course management system
* Function: The component accomplishes collecting data inputs from the user and relaying this to the logic tier and displaying requested data on courses.
* Subordinates:
* Dependencies: Uses style element from the Bootstrap library to help with placement of information in the web window. Uses the WTforms elements from the Flask library to help display information from the logic tier on the web page.
* Interface: Interacts with the logic tier through the Flask library’s module WTforms. This is accomplished through various GET and POST requests made via the logic tier.
* Resources: External location to host static and dynamic web files.
* Processing: No algorithms in play for the UI, handling of authentication is done through the Flask library and its flask\_login module.
* Data: Data is represented by their accompanying courses. A course can have multiple fields of data describing it, all of which comes from the database through the logic tier. Data is transferred internally through a JSON format like:



and communicated through a REST api

## Logic Tier

* Identification: The Logic Tier / Business Rules
* Type: A Python web server consisting of multiple files containing classes for the database models, the various web routes and the forms which will be used.
* Purpose: Processes requests from the user interface tier and makes logic decisions on how to handle such input.
* Function: This component accomplishes passing requested data from the database tier to the user at the user interface tier all while correctly interpreting the data and ensuring all requests are properly authenticated.
* Subordinates:
* Dependencies: Uses multiple modules of the Flask python library. One of which is WTforms to help display data onto static html pages at the front end. Also uses the Flask SQLAlchemy module to facilitate communication to the database tier. Utilizes the Flask login manager module to authenticate users when trying to access parts of the system.
* Interface: Interfaces with the User Interface through WTforms and Flask, enabling data to be sent from the logic tier to the user interface and allowing the logic tier to redirect users to other pages on the UI. Interfaces with the database tier through SQL queries and the SQLAlchemy Flask library.
* Resources: External location to host the web app files and ensure they are always running.
* Processing: Handles errors in authentication and connection through logging that can be seen by the person overviewing the server. Initialized through a setup script that can either wipe the database and start fresh for new clients or reinitialize from its previous state in the case the program stops running
* Data: Data is represented in a JSON scheme and transferred to the UI tier via REST api. The data is retrieved from the database tier via SQL queries and SQLAlchemy method calls.

## Database Tier

* Identification: The database tier / data store
* Type: A SQLite3 database
* Purpose: Stores a university’s information on courses for retrieval from the course registration management system.
* Function: The component keeps information on university courses safe in a central location and oversees requests to access this information.
* Subordinates:
* Dependencies: Uses nothing else but what SQLite3 has to offer.
* Interface: Interfaces with the logic tier through the Flask module SQLAlchemy. The logic tier invokes SQLAlchemy methods to access, modify and retrieve data from the SQLite3 database.
* Resources: A machine with enough memory to hold a large collection of data, redundant power supplies incase of an outage, and easily accessible to an admin of the institution or of the company.
* Processing:
* Data: The database tier is where the data is stored in a straightforward structure like how the rest of the application interprets the data. Each user is stored in a user table, courses are stored in course tables and all the attributes pertaining to these two items are columns within these tables. This allows for easy access using SQL queries.