Course Manager

Benjamin Choi
BSMS Candidate
December 2019
Department of CSE
Washington University
St. Louis, MO
Email: benjaminchoi@wustl.edu

Currently, TA checkout procedures for classes such as CSE 131 and 247 rely on TAs entering secure information onto students' browsers. This is an obvious security concern that should be fixed. With my master's project, I wanted to create a secure system for TA checkouts that rely on QR codes. The system will have a cross-platform mobile app and a web frontend. After an instructor invites students to a course, students will be granted a unique QR code for every assignment. TAs and instructors will then be able to check out students' assignments using either the mobile app or website. In this way, students never gain access to any information beyond their unique QR codes. The system would also be able to track which TAs checked out which students, helping to avoid possible conflict-of-interest issues.

Nomenclature

CRUD Create, Retrieve, Update, Destroy
DRF Django Rest Framework
DRY Don't Repeat Yourself
REST API REpresentational State Transfer Application Programming Interface
SPA Single Page App(s)

1 Motivation

Although the project was inspired by a need in the CSE department, I wanted to pursue implementation in a way that would also serve as an educational experience and an opportunity for me to expand my knowledge of industry-standard tools. As a result, the project was implemented using Django, React, and React Native. DRF was used as a way for the Django backend to communicate with the React and React Native frontends. The use of React and React Native makes the project both desktop and mobile-friendly, while the use of Django and DRF allows the two frontends to access the same data. While I did have limited exposure to Django, DRF, React, and React Native prior to this project, I saw this as an excellent opportunity to expand my proficiency with these tools to the point where I would feel comfortable using them for future works or working in industry.

As explained in the abstract, a big focal point of the project was security. Thus, configuring the REST API to only allow access to relevant parties served as one of the biggest hurdles of the project and how to do so became one of the most useful things I learned over the course of my work.

2 Initial Setup

To properly demo the work, I set up a publicly-accessible demo website as a subdomain on my personal website, hosted on a DigitalOcean Droplet that serves up content using Postgres, Gunicorn, and Nginx.

Before doing anything else, I linked my website to the Droplet by configuring the DNS settings of the domain to use DigitalOcean's name servers. I then configured the networking settings on DigitalOcean to point the subdomain linked to the demo to the IP address associated with my Droplet. [1]

To set up the Droplet, I created a firewall using UFW and allowed OpenSSH and Nginx access. I then installed Python, Postgres, and Nginx onto my machine. On Postgres, I created a new "coursemanager" database and a new "django" user, giving the "django" user all privileges on the "coursemanager" database. In order to set up Gunicorn and the Django project itself, I created a new Python environment for the project using virtualenv and managed all of the needed dependencies with a requirements.txt file (including django, gunicorn, and psycopg2). From there, I initialized a new Django project, added my domain and the Droplet's IP address to the list of allowed hosts, changed the database to use the "coursemanager" Postgres database, pointed the STATIC and MEDIA directories to be inside the base directory, migrated, collected the initial static content, and created an initial superuser. After creating the Django project, I initialized Gunicorn by setting up the necessary gunicorn.socket and gunicorn.service files, making sure to point them to the correct working directory and project path. Finally, I configured Nginx by creating an nginx file for the project and pointing it to my domain and project directory. [2]

To expedite this process for future users, I created a setup script and wrote instructions on how to use it on the project's Github README (see Documentation and Project Resources).

3 Design

As a programmer, the ability to make forward-thinking design decisions on-the-fly is an invaluable skill and one that I felt was strengthened over the course of the project. In this section, I will detail how Course Manager is designed and why I made these design choices.

3.1 Django

3.1.1 Framework Choice

Due to the university's investment in Canvas, it would have been possible to sidestep the creation of a backend and instead serve up content using Canvas's REST API. However, I decided to implement my own backend for two reasons: to learn how to create a backend capable of complex, unique use-cases and to make Course Manager a potentially useful tool outside the scope of WUSTL specifically. Had I not implemented the backend myself, I would have learned significantly less in terms of sensible model design, the creation of a REST API, handling requests and permissions, and combining a frontend with a backend. Additionally, without its own backend, Course Manager would be a completely useless tool for any institution not using Canvas.

As mentioned before, while the project started from a need in the CSE department, I also wanted to treat it as an opportunity to expand my knowledge of full-stack development and the various tools involved in full-stack development, including the creation of a backend. While the decision to implement my own backend was an enriching educational experience and learning opportunity, it did come at the loss of compatibility with Canvas. Steps to remedy this issue is discussed in Future Work.

As for the choice of Django specifically, I chose it due to its wide adoption in industry, its use of and my familiarity with Python, and my prior experiences with the framework.

3.1.2 Models

Course Manager is made up of six models that define how the project's data is stored in Postgres. The following details the names of the models, their relevant fields, and the data type of each field. Note that the first field detailed acts as the primary key of the model and references to the model will be represented in the database by its primary key.

1. User: {email: email address, first_name: string, last_name: string, is_staff: bool, is_active: bool}
2. RegisterInvitation: {id: int, recipient: User, token: string}
3. Course: {id: int, course_id: string, title: string, instructors: [User], tas: [User], students: [User]}
4. AssignmentGroup: {id: int, course: Course, group: AssignmentGroup, title: string, points: int, due_date: datetime}
5. Assignment: {id: int, course: Course, group: AssignmentGroup, title: string, points: int, due_date: datetime}
6. StudentAssignment: {id: int, assignment: Assignment, student: User, qr_code: string, completed: bool, points_earned: float, timestamp: datetime, is_late: bool, grader: User, comment: string}

Although Django comes with a default implementation of the User model, I decided to use a custom User model (a choice that required a great deal of code refactoring). The chief differences between the custom User model and the vanilla User model are: the custom User model lacks an auto-incrementing numeric id field, a unique username field, and requires the email field to be unique. Although Django makes it comparably difficult to use a custom User model, I chose to use one in order to get around the issues related to course invitations. Due to the nature of Course Manager, it made most sense for instructors to be able to invite other instructors, TAs, or students to courses via email (the same way services like Bitbucket invite users to join a repo as a collaborator). However, this presented a natural challenge as it is impossible to add a user to a course's instructors, tas, or students field without the User already being in the system. When thinking of ways to get around this issue, I came up with a couple viable strategies:

1. Instructors should prompt students to sign up for Course Manager, providing a key to enroll in the course (similar to Piazza)
2. The instructors, tas, and students fields should not represent a list of User models, but should instead represent a list of emails which can be used later to retrieve User models once invitees register
3. When instructors add an email to the instructors, tas, or students field, it should create a new User with an unusable password with the is_active flag set to false, and then send the user a register invitation that will set up a proper password and set the is_active flag to true

Although all are viable strategies, I decided to go with the third option. As Course Manager would presumably have a very specific use-case, it made little sense for random parties to be able to register for an account. Thus, I decided that users should only be able to register when invited to do so by an instructor. Between options two and three, I decided to go with option three as it provided greater ease when working with nested serializers for the REST API. This choice necessitated the use of the custom User model, as it would have been impossible to create a vanilla Django user model with an email alone, instead necessitating the creation of a unique username. [3]

Otherwise, most of the other models are fairly self-explanatory. The RegisterInvitation carries a one-to-one relationship with a User model, and allows the user to formally set up an active account once invited to use Course

Manager with an auto-generated token. The Course model represents a course, the AssignmentGroup model represents a category of assignments (e.g. Labs), while an Assignment model represents a specific Assignment within an AssignmentGroup (e.g. Lab0). Although both AssignmentGroup and Assignment have points values, the instructor can have the total points values of all Assignment objects within an AssignmentGroup surpass the total points for the AssignmentGroup. The thought behind this behavior came from CSE 131 extensions, in which students can choose to do as many extensions as they want, but will only receive full credit if they surpass a given points threshold. Finally, the StudentAssignment object is unique for any given User and Assignment combination, and represents that student's unique attempt at completing the assignment. Critically, it holds the qr_code field that will be used to check them out. As all QR codes ultimately represent string values, I decided to simply store the QR code as a string and render it on the frontend (rather than storing an image of the QR code on the server and making the qr_code field point to the file).

3.2 DRF and React

3.2.1 Framework Choice

It is important to acknowledge why I chose to use React in the first place. After all, it is possible to meet all frontend needs without DRF or React, relying instead on Jinja.

First, the need for DRF becomes clear when thinking about the mobile component of the project. While it would be possible to create a mobile-friendly website using Jinja and a CSS tool like Bootstrap, creating a mobile app (something I aimed to do for Course Manager) necessitates the use of DRF to serve up data from a central source of truth. Additionally, creating a REST API for the backend allows for greater extensibility, making the project much more future-proof and allowing for generally more flexible use-cases (such as compatibility with third-party scripts).

Second, I chose to use React instead of Jinja for two reasons: to put all of the CRUD logic in the REST API and to exploit the DRY design philosophy allowed by React's use of components. As mentioned before, I decided to make the project as extensible as possible by creating a REST API. However, creating a well-functioning REST API, while duplicating all of the CRUD logic in vanilla Django to be compatible with Jinja's server-side template rendering scheme, seemed like a poor design choice, breaking DRY philosophy and increasing development time. Thus, it made little sense to use Jinja while also committing to the creation of a REST API. Using React not only allows the frontend to serve up content using a REST API, it also allows developers to reuse large chunks of the frontend with its use of components. Overall, using Django and React, with DRF as the bridge between the two, seemed like the best design decision to achieve my goals.

When creating a web application using Django, DRF, and React, there are three commonly used design patterns:

1. React in its own "frontend" Django app, in which a single HTML template is loaded and React manages the frontend
2. DRF as a standalone API and React as a standalone SPA
3. Using mini React apps inside Django templates

Ultimately, I decided to go with the first option for a couple of reasons. As I wanted to do as much of the frontend using React as possible, I ruled out option three right away (this would also make working with React Native easier as React and React Native share much of the same syntax and flavor). Of the two remaining options, option two seemed best for my use case as it kept React closer to Django, better allowed for an interface with a lot of user interactions/requests, and is better suited for an app-like website. [4]

3.2.2 Serializers

Due to the behavior I wanted in Course Manager, it became necessary to highly customize model create and update behaviors. When working with DRF, it is generally considered best-practice to put as much of that logic into the serializers as possible. As a result, many of the serializers for the various models explained in Models required significant code refactoring, something I had little-to-no prior experience with in DRF.

For the User model, it was important to keep in mind that the only time a new User would be created is when instructors add emails to a course's instructors, tas, or students field. Thus, I needed to create a special RegisterSerializer that would only be called when a course's User-related fields were created or updated. However, whenever a new user is created using the RegisterSerializer, it would also need to create a RegisterInvitation so that invitees can receive emails prompting them to activate their accounts. As a result, the create method of the RegisterSerializer not only needed to create a new inactive User, it also needed to create a RegisterInvitation using the RegisterInvitationSerializer. The RegisterInvitationSerializer, on the other hand, also needed a modified create method that would not only create a RegisterInvitation model, but would also send an invitation email to the recipient with the recipient's token embedded in the invitation link. Additionally, as it would be important for students to be able to activate their accounts, I also needed to create a TokenRegisterSerializer that allowed inauthenticated users to update their accounts to be active using their invitation links.

Once it was guaranteed that all of the User objects in a course's user-related fields existed in the database (inactive or not), the CourseSerializer also needed to ensure that whenever new students are added to a course (either during a Course model's creation or modification), it also creates unique StudentAssignment objects for each assignment in the course. Conversely, it was also important for the creation/modification of a new Assignment to result in a corresponding change in the StudentAssignment model for each student in the course. In the case of an update to the Assignment model, it was important to ensure that changing an assignment's possible points value or due date would not result in students getting unfairly credited or penalized, thus the update method of the Assignment serializer ensures that

the points and due date fields scale to correspond with the new values (e.g. if a student received a 4/5 on an assignment and the assignment's points value is changed from 5 to 10, the student's earned points changes from 4 to 8). Some additional interactions between the models take place in the API Views rather than the serializer, the reasoning behind which can be found in the API Views section.

3.2.3 API Views

In DRF, API Views act as the endpoints for communication between clients and the server. The API Views are thus what call the serializers. As mentioned before, it is generally best practice to leave as much of the interdependent CRUD logic in the serializers as possible, but it did become necessary to move that logic to the views in very specific circumstances.

Chiefly, in the views related to the Course model, it was necessary to move the automated creation of inactive users to the view. This is mainly because before a model object can be created or updated by a serializer, it first performs a series of validity checks. As trying to add nonexistent users to a user-related field fails these validity checks, it became necessary to move the logic for the creation of inactive users to the view. Thus, the view for the Course model is actually what calls the RegisterSerializer, which in turn starts a chain of events.

Otherwise, it was only necessary to change the default behavior of DRF's views when responses needed to be customized. For example, when a user logs in, the view associated with logging in also returns all of the courses that user is a part of as an instructor, TA, or student. These modified responses are meant to increase efficiency by decreasing the need for a series of calls to the REST API, instead relaying all needed information in one request/response.

3.2.4 Policies

As mentioned in the abstract and the Motivation section, security was a big focal point of the project (both for practical and educational reasons). While it would be fairly easy to restrict the frontend to show features by user status (such as whether they were admins, instructors, TAs, or students), it would leave the REST API vulnerable to possible attack unless the API itself was restricted by user status. While it is fairly easy to restrict the API by whether the user is logged in or is an admin, it is not simple to restrict views according to qualifiers like whether a requester is in a Course model's "instructors" field. Thus, it became necessary to use Rest Access Policy, a third-party app built for DRF meant to allow for extensible permissions schemes.

When working with policies, the biggest difficulty came from differentiating between "detailed" and "non-detailed" requests. While detailed requests work on a specific instance of a model object, non-detailed requests work on a group of model objects. In CRUD, update, and destruction are all explicitly detailed requests. However, retrieval can be detailed or non-detailed. Additionally, although creation returns a single object like a detailed request response, as the object does not yet exist in the database the request is formatted like a non-detailed request. As a result, the policy checks need to keep this distinction in mind as detailed requests are able to access a specific model object and can check for the user's relationship to that object, whereas a non-detailed request cannot.

For added security, the way models are serialized also differ according to a user's position within a course. For example, instructors and TAs receive a list of all students for a given course, but students only receive limited information about instructors and TAs, and zero information about other students in the course. Overall, the REST API is extremely robust in terms of permissions, only allowing users to access features relevant to them and behaving as expected. One caveat is that users with staff status (admins) have blanket all-around access to every point of the REST API.

3.2.5 Frontend

In modern web development, high design standards demand that websites should be intuitive, mobile-friendly, and beautiful. In that pursuit, Bootstrap has become a standard tool in any frontend programmer's toolkit, and something I chose to integrate heavily into the frontend. React even makes this extremely easy to do with React Bootstrap, which turns vanilla Bootstrap CSS elements into easily configurable React components. Out of the wide array of Bootstrap themes, I chose to go with the Darkly theme.

While the Django backend acts as the central source of truth for both the React and React Native frontends, the components of React need an internal source of truth as well (after all, components should not continuously query the database for the same information). Out-of-the-box React components lack the capacity to share state and prop elements intuitively, necessitating the use of tools like redux. Redux integrates with React to provide a central state shared between all connected React components, allowing the frontend to maintain an internal source of truth. Although requiring some more configuration upfront, Redux's use is nearly unavoidable for all large-scale, query-based React websites.

Finally, in keeping with the topics discussed in the Policies section, I also restricted the frontend's views to accurately reflect the content served to the user by the REST API. For example, it makes no sense for TAs and students to see the option to edit or delete courses, assignment groups, and assignments if they do not have the permissions to do so. Thus, the frontend was designed to keep in-line with the permissions scheme outlined in the REST API's polices. See Fig. 1, 2, 3, & 4 for reference.

3.3 React Native

3.3.1 Framework Choice

Although it is possible to create mobile-friendly websites using CSS tools like Bootstrap, I decided to create a mobile app for the same reason that most tech companies choose to create mobile apps instead of relying on a mobile-friedly website. While users generally dislike having to install an application for one-off use, they prefer to interact

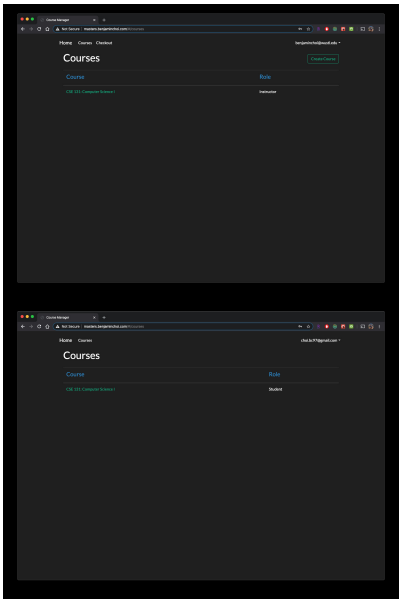

Fig. 1. A side-by-side comparison of an admin's homepage and a non-admin's homepage (from top to bottom)

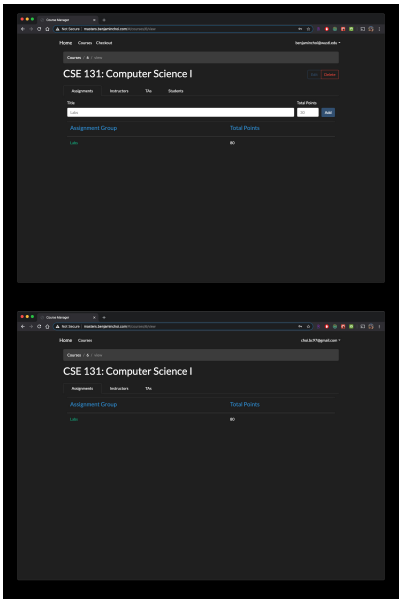

Fig. 2. A side-by-side comparison of an instructor's course page and a student's course page (from top to bottom)

with an app when they need to use it frequently. As TAs and instructors would presumably need to check out students at least twice a week, I felt that most users would prefer an app. Additionally, due to Apple's restriction of the iPhone, it is impossible to access the camera (something needed to make the QR code scanning checkout as quick as possible) without using Apple's Safari browser. Additionally, no matter how mobile-friendly the website is, a mobile app simply offers greater ease of use when working on mobile phones. Finally, just as I felt that implementing the backend would be an excellent opportunity to expand my knowledge, I felt the same way about the creation of the mobile app.

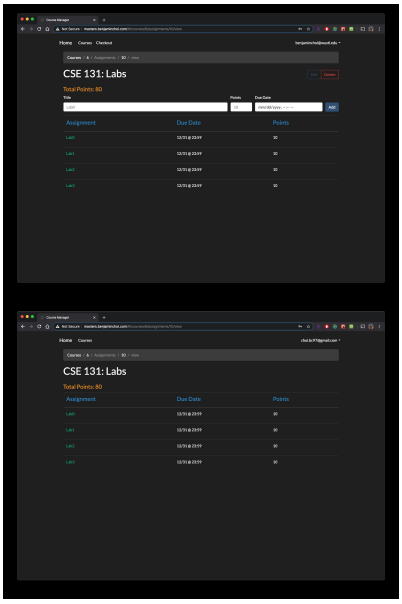

Fig. 3. A side-by-side comparison of an instructor's assignment group page and a student's assignment group page (from top to bottom)

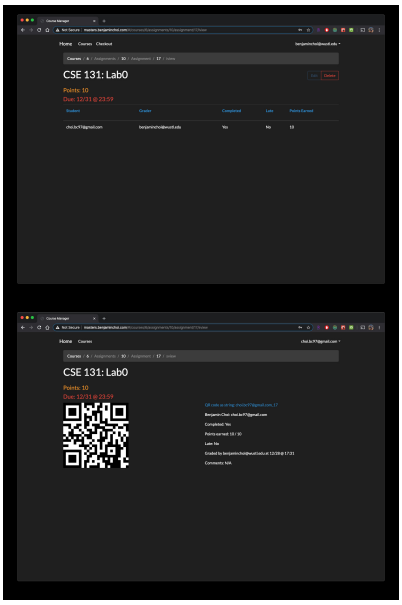

Fig. 4. A side-by-side comparison of an instructor's assignment page and a student's assignment page (from top to bottom)

As for the choice of React Native specifically, it seemed like a no-brainer after choosing React for my web frontend due to their common design philosophies and general syntax.

3.3.2 Navigation Flow

When designing the app, it was important to keep in mind what the intended use case would be. For this project, it was to allow instructors and TAs to make use of their phones' cameras in order to check out students' QR codes as quickly as possible. Thus, the app only needed to cater to instructors

and TAs, without needing a complicated navigation flow. As a result, the app can be broken down to just 4 screens connected by 2 navigators: a loading screen to see if the user is signed in already, a sign in screen to allow the user to sign in, a scanner screen to allow the user to scan QR codes, and a checkout screen for the user to assign students points and comments. The scanner and checkout screens are connected by a StackNavigator, which is in turn connected to the loading and signing screens by a SwitchNavigator. Much like the React frontend, the React Native frontend also makes use of redux to create a central source of truth shared between all React components. The app is simple and purposefully so (see Fig. 5).

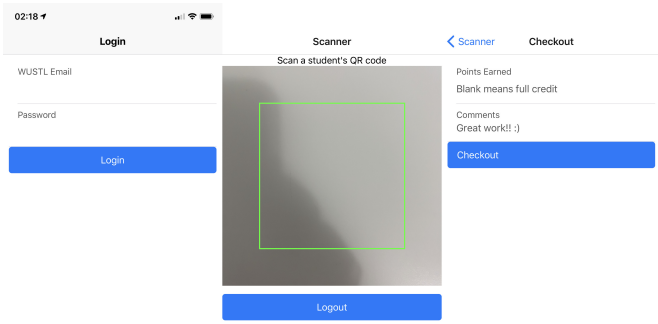

Fig. 5. The login screen, scanner screen, and checkout screen on iOS (from left to right)

4 Documentation

As stated in the Initial Setup section, I thoroughly documented how to setup, use, and make changes to the project on the Gibhub repo's README (see "Github repo" on Project Resources). This documentation also includes through instructions on how to access and work with the REST API created by DRF. As a result, an explanation of these steps and instructions are omitted on this project report and can instead be found on the Github repo.

5 Lessons Learned

In many ways, I had to learn much of what I had to do for this project from scratch. This project was the first time that I had to learn how to adjust a domain registrar's DNS settings to point to a cloud provider's name servers, set up a subdomain, get a cloud server up and running, manage Gunicorn and Nginx, customize Django's and DRF's standard behaviors, set up a robust permissions scheme for a REST API, and build a cross-platform mobile/web frontend. Although I did have limited experience with Django, DRF, React, and React Native from previous projects, I had never interacted with it with the same amount of depth as I did for this project.

Not only do I feel like I have strengthened my full-stack developments skills, this project was also a lesson in intelligent design and product management. For example, during the first semester of the project, I feel in retrospect that I spent too much time debating the possible routes to go down rather than committing to a single path and sticking with it. The uncertainty at the start had dire consequences in terms of product management. Initially, I felt it was okay to thoroughly explore all of the possible options for what direction the project should go down as I grossly underestimated how much time various tasks would take during development. Originally, I had intended to include two or three additional features that never made it into Course Manager.

Another mistake I made is that I decided to tackle the React Native frontend before any other portion of the project as, at the time, that was the framework I was most familiar with. By the time I finished the Django and React portion of the project, however, I discovered that much of the work that I did for the React Native frontend would not be ready for the project presentation as React Native overhauled their dependency scheme in a huge update, which left my React Native project broken. For future projects, I now know that the backend and frontend should either be tackled concurrently or the backend should be finished first, in order to avoid issues like this in the future.

6 Future Work

As mentioned previously in the justification for using Django, the decision to implement my own backend came at the cost of using Canvas for the backend. In the future, independent of this project, I plan to incorporate an option into Course Manager that would allow instructors to instead use Canvas's REST API as opposed to the one I built with DRF.

For the future, I also see three tasks that would make Course Manager much more useful for WUSTL (though not necessarily for other institutions).

1. Configuring Course Manager to work with a tool like Jenkins in order to check if students committed and passed all of the tests associated with an assignment before allowing a TA or instructor to check them out
2. Incorporating the features explored in other course logistics Master's projects, such as this TA office hours tool built by WUSTL alum Kateryna Kononenko.
3. Integrating Course Manager with WUSTL's Connect API to push user management to the university's existing architecture, which would add the bonus benefit of allowing TAs and instructors to visually check the identity of students with their university picture

These are all additions that I would like to pursue, but would encourage any other WUSTL students to attempt. While I feel like this project became an excellent educational opportunity, it still needs some work before it becomes an in-

valuable tool for the CSE department, although I believe it is nearly there.

7 Conclusions

As stated in my abstract, I wanted to build a secure, cross-platform TA checkout system based on QR codes. In terms of what I set out to do, as stated on my original project proposal, I achieved the tasks I outlined for myself. As a result of my efforts, I feel I have grown considerably as a full-stack developer, product manager, and product designer. Despite this, I feel that Course Manager still has a lot of room for growth that would make it increasingly more useful for the university.

8 Project Resources

The following are the links to all of the work related to this project:

Project demo: http://masters.benjaminchoi.com/
Github repo: https://github.com/choibc97/coursemanager
YouTube clip of a QR checkout demo: https://youtu.be/disuoQwKahA
Project presentation: https://docs.google.com/presentation/d/12zC06G4hU9abvGivPs0GsLLakLDVvjK9q8Ief1xmx_w/edit?usp=sharing

Acknowledgements

Thank you to Professor Dennis Cosgrove for acting as my project advisor, mentor, and friend. Thank you to Professor Ron Cytron for being my major advisor, a committee member, and for introducing me to the world of Computer Science in CSE 131. Thank you to Professor Todd Sproull for being a committee member and for the lessons in full-stack development. Thank you to all of the members of the WUSTL CSE Department for your time and instruction, I have learned a lot. Lastly, thank you, whoever is reading this, for taking the time to examine my work.

References

[1] Ellingwood, J. Initial server setup with ubuntu 18.04. https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-18-04.

[2] Ellingwood, J. How to set up django with postgres, nginx, and gunicorn on ubuntu 18.04. https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-18-04.

[3] Herman, M. Creating a custom user model in django. https://testdriven.io/blog/django-custom-user-model/.

[4] Gagliardi, V. Tutorial: Django rest with react (django 3 and a sprinkle of testing). https://www.valentinog.com/blog/drf/#Django_REST_with_React_Django_and_React_together.