Developer Documentation

ProofChecker - SE691

Rakhfa Amin, Thoman Andrews, Kersley Jatto, Aiasha Sattar, Colton Shoenberger

Purpose of this Document:

The intention of this document is to provide information of value to future developers of the ProofChecker system. The system is fairly sophisticated, consisting of over 10,000 lines of code at the time of writing this document. It could be easy to get lost in the complexity of the code without some guidance. We hope this document will make it easier for developers looking to extend and maintain the system.

Introduction to Django:

Perhaps the first important thing to know about the project is that it is a web application built using the Django framework. Django is the most popular Python web framework and it is extensively documented (https://www.djangoproject.com/). The Django site offers plenty of information to get you started, including a seven-part tutorial that helps get you up-to-speed on the basics fast (https://docs.djangoproject.com/en/4.0/intro/tutorial01/). For those new to Django, we highly recommend completing the tutorial. It should take approximately two hours of your time, and you will be much better equipped to understand our system after completing the tutorial.

Model-View-Template (MTV) Pattern

Django embraces the popular Model-View-Controller (MVC) pattern for web development, though with slightly different names. In Django, this pattern is called Model-Template-View (MTV).

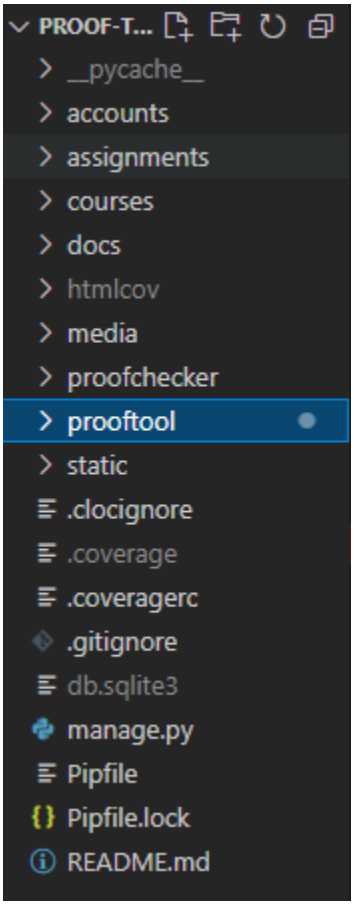
- *Models* - A model is a representation of an object that is persisted to a database. Models for a Django "app" are defined in Python files named `models.py`. We will explain the concept of a Django "app" shortly. Documentation for Django models can be found here: https://docs.djangoproject.com/en/4.0/topics/db/models/
- *Templates* - Templates in Django represent what is referred to as a "View" in the MVC pattern, which can cause some confusion. In short, a template is an HTML file. It represents what the user sees on their computer screen. Within the directory for each app is a `templates` directory, which is where all the html files are located.

Documentation for Django templates can be found here: https://docs.djangoproject.com/en/4.0/topics/templates/

- *Views* - Views in Django represent what is referred to as a "Controller" in the MVC pattern, which, again, can cause some confusion. In short, a View represents the logic that controls the interactions between Templates and Models. To borrow from the Django documentation itself, "a view function, or view for short, is a Python function that takes a web request and returns a web response". Views for each Django app are found in the Python file named `views.py` for each Documentation for Django views can be found here: https://docs.djangoproject.com/en/4.0/topics/http/views/

Directory Structure

If you're looking to maintain the system, we want it to be easy for you to find the files you're looking for, and if you're looking to extend the system, we want it to be easy for you to know where to add code. Thus, this section seeks to explain the file structure of the system.



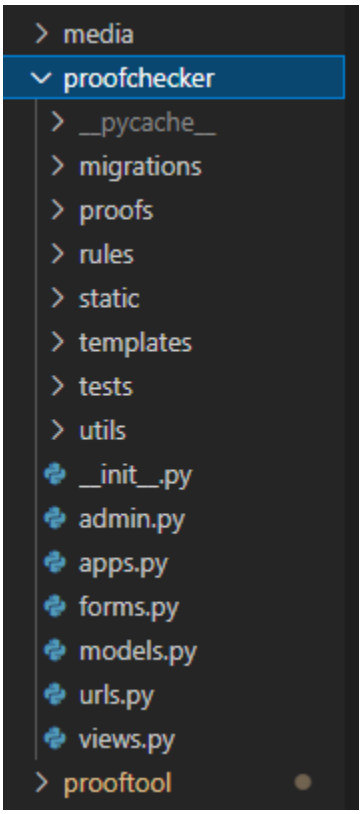
The first thing to know is that the system, as it is known to Django, is named `prooftool`. It could be renamed, but it would probably be more hassle than it's worth. The `prooftool` directory contains a few files that control the overall system:

- `settings.py` - This is probably the most important file in this directory. As its name suggests, it defines settings for the system, as well as important configurations with respect to deploying the system in an environment. Documentation for Django settings can be found here: https://docs.djangoproject.com/en/4.0/topics/settings/
- `urls.py` - This file contains definitions of paths used for URL routing. Each Django app will also contain its own `urls.py` file with additional paths. Documentation for Django URLs can be found here: https://docs.djangoproject.com/en/4.0/topics/http/urls/
- `asgi.py` and `wsgi.py` - This file contains code pertaining to web-server gateway interfaces. It is pre-written by Django, and it is unlikely you will find any reason to amend these files.

Django Apps

It is important to understand the concept of an “app” in Django. If you are familiar with the C4 model for software architecture (https://c4model.com/), a Django “app” is essentially a “Component” of the system. As it is defined in the Django documentation, “the term application describes a Python package that provides some set of features.”

There are currently four apps (or components) in our system: `accounts`, `assignments`, `courses`, and `proofchecker`. You will find a directory for each. We will begin with an explanation of the `proofchecker` app, since it is the most significant component of the system.



`proofchecker`

`proofchecker` represents the tool that users interact with to create and verify proofs. Inside this directory you will notice a few familiar things that we have already explained.

- There is a `models.py` file that defines objects persisted to the database. For example, a “Proof” has a model, as well as a “ProofLine”.
- There is a `views.py` file, which handles HTTP requests and responses. For example, when a user clicks the “Check Proof”, an HTTP request is sent to the server to verify the proof, and an HTTP response is returned that includes the results of the proof verification.
- There is a `urls.py` file, which defines URL paths
- There is a `templates` directory, which contains HTML files

Next, we will explain the purpose of some of the other files and directories within the `proofchecker` app.

- `migrations:` The `migrations` directory contains files with instructions Django uses to create your database. These files are written automatically by Django, so there is no need to edit any of the files in this directory. We will explain migrations in Django in

more detail later. Documentation on migrations in Django can be found here: https://docs.djangoproject.com/en/4.0/topics/migrations/

- `proofs`: The `proofs` directory contains three important files:
    - `proofchecker.py`: This file has two important functions:
        - `verify_proof()`: This function is called when the user clicks the "Check Proof" button. (More specifically, it is called within the `views.py` file when a user submits a HTTP request that contains `check_proof`). It returns a `ProofResponse` object, which is explained shortly.
        - `verify_rule()`: Called within `verify_proof()` for each line in the proof, this method verifies if the rule provided on a ProofLine is applied appropriately.
    - `proofobjects.py`: This file defines three objects, described briefly below. You may note that two of these objects are also defined in models.py. The reason for representing them here is that Django Models use Django Fields to represent attributes of the object, whereas the classes in this file use primitive Python data types, which are easier to manipulate for proof validation. Documentation for Django Fields can be found here: https://docs.djangoproject.com/en/4.0/ref/models/fields/
        - `ProofObj`: A representation of a proof
        - `ProofLineObj`: A representation of a ProofLine
        - `ProofResponse`: A representation of the response returned to a user when they request to check (verify) a proof for completeness and correctness.
    - `proofutils.py`: This file contains "getters" and "setters" as well as other custom functions that assist with the validation of proofs and rules.
- `rules`: The `rules` directory contains a file for each rule that a user might apply within a proof. Currently, this includes every basic and derived rule for Truth Functional Logic (TFL) and First Order Logic (FOL). There are two special files in this directory that are worth pointing out:
    - `rule.py`: Every rule in the system subclasses this abstract `Rule` class. Each rule is required to implement a `verify()` method, which is called during proof validation when the `proofchecker` is attempting to verify that a rule is applied appropriately on a proofline. It should be noted that every rule should also

provide a `symbols` attribute that defines symbols used to represent the rule within a system (e.g. "∧I" are the symbols for the rule Conjunction Introduction)

- `rulechecker.py`: Defines a `get_rule()` method that is called to determine which rule is being applied on a line (by matching the symbols provided by the user to the symbols attribute of a rule)

- `static`: Contains "static" files. There are three sub-directories, which are fairly self-explanatory:
  - `css`: Contains CSS files, used for styling web pages
  - `images`: Contains image files
  - `js`: Contains JavaScript files, which handle behavior and functionality of web pages. There are a couple important files worth mentioning here:
    - `proof_checker.js`: This file defines functions that control the behavior of many of the buttons in the ProofChecker tool that users interact with. For example, the functionality pertaining to adding and deleting lines, creating subproofs, and automatic line numbering is all defined within this file.
    - `right_menu_navigation.js`: Controls behavior that relates to the right-hand menu.
- `tests`: Contains test files. We will return to a discussion of how to run tests in Django later. Documentation for testing in Django can be found here: https://docs.djangoproject.com/en/4.0/topics/testing/
- `utils`: Contains "utility" files. Many of these files are quite important to the integrity of the system, so we explain some of them below:
  - `ply`: Python Lex & Yacc (PLY) (https://www.dabeaz.com/ply/) is a third-party tool that our system relies upon significantly in order to parse expressions provided by users in ProofLines. Documentation for PLY can be found here: https://ply.readthedocs.io/en/latest/index.html. It should be noted that none of the files within the `ply` directory should be altered. They were not written by us, but by the creators of PLY. There are two concepts that must be understood to grasp how PLY works:
    - Lexer: A lexer, or a tokenizer, takes an input string and outputs a sequence of "tokens". For example, the file tfllexer.py defines all the legal tokens within the syntax of TFL (i.e. `VAR, BOOL, AND, OR, NOT, IMPLIES, IFF, LPAREN, RPAREN`). Similarly, the file `follexer.py` defines all the legal tokens within the syntax of FOL. `numlexer.py` is

used for handling line numbers within proofs, which, due to the possibility of having more than one decimal point, cannot simply be represented as floating point numbers in Python. If a character in the input string does not match a token in the lexer, it will output an error.

- Parser: A parser takes a sequence of Tokens provided by a lexer and matches them to a Backus Naur Form (BNF) specification of grammar. If the parser is unable to match a series of tokens provided by the lexer to a pattern defined within your BNF specification, it will output an error. If it is able to successfully match all the tokens to the BNF specification, it will act according to the instructions you provide the parser. The parsers in `tflparser.py` and `folparser.py` create a binary-tree representation of the input expression (string). The `numparser.py` file simply calculates the depth of a line number (e.g. line "2.1.2" has a depth of 3).

- `binarytree.py`: As mentioned above, The TFL and FOL parsers in our system produce binary tree representations of the input expression on a ProofLine. The binary tree class is defined in this file. Note that it is not necessary that a parser produce a binary tree representation. A parser can behave according to any instructions you provide it. If you wish to extend the system with a new form of logic or mathematical reasoning where expressions are not well-represented as a binary tree, you could create a new lexer and parser that parses expressions into another type of data structure.
- `constants.py:` Contains a class of constant values referenced within various parts of the system.

- `admin.py`: This file contains instructions for how to display information on the Admin page of our website. We will explain the Admin page in more detail later on. Documentation on the Django Admin site can be found here: https://docs.djangoproject.com/en/4.0/ref/contrib/admin/
- `forms.py`: Django Forms essentially instruct the system how to create an HTML form from a Model object. An HTML form, as explained in the Django documentation, "allows a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server". Within our system, a "Proof" has a form (with the fields "Name", "Rules", "Premises" and "Conclusion"), and a ProofLine has a form (with the fields "Line Number", "Expression" and "Rule"). Documentation on Django Forms can be found here: https://docs.djangoproject.com/en/4.0/topics/forms/

That wraps up our discussion of the `proofchecker` app, the most sophisticated app within the system. As mentioned previously, there are three other apps in the system: `accounts`, `assignments`, and `courses`. If you open up the directory for any of these apps, you will quickly notice a fairly similar structure to the one we just reviewed with `proofchecker`. Each app contains the files `models.py`, `urls.py`, and `views.py`, as well as a `templates` directory. To provide a brief explanation of each app:

- The `accounts` app handles user accounts, including the new user registration process, and the sign-in process for existing users.
- The `courses` app provides the ability for Instructors to create Courses in the system. Instructors can add Student users to their courses, and Students can join available courses.
- The `assignments` app provides Instructors the ability to create Assignments in the system that they can distribute to students who are enrolled in their courses. Students can then view their Assignments page to begin working on Assignments for courses they are enrolled in.

That concludes our discussion of the four main apps that comprise the `prooftool` system. Next, we turn to an explanation of some of the remaining files and directories that may be of interest.

pipenv

Pipenv (https://pipenv.pypa.io/en/latest/) is a packaging solution for the system. It stores a list of all the project dependencies in a file named `pipfile`. To install a virtual environment with all your project dependencies, navigate to the top-level directory of proof-tool and run the command "`pipenv install`" on the terminal. To activate the virtual environment, run "`pipenv shell`". If you wish to install an additional third-party dependency in the project, run the command "`pipenv install {name-of-dependency-here}`". Additional documentation for pipenv can be found at the link above. Note that you will want to have activated the virtual environment whenever you are running python commands for the project, such as `migrate`, `test`, and `runserver` (explained in the next section).

manage.py

manage.py is a file that is automatically created in each Django project. It allows you to run Django's command-line utilities. If you completed the Django tutorial, you should already know these. If not, get to know these commands, as you will find yourself needing to run them often. Also, as mentioned above, make sure to have activated your virtual environment with `pipenv shell` prior to running these commands.

- `makemigrations`: This command creates migration files, which Django uses to map your Models to database tables. To run this command, type '`python manage.py makemigrations`' on the command line (from the top-level directory of proof-tool, while your virtual environment is active)
- `migrate`: This command actually updates your database tables. To run this command, type '`python manage.py migrate`'. Additional documentation about Django migrations can be found here: https://docs.djangoproject.com/en/4.0/topics/migrations/
- `test`: As you may have guessed, this command runs all of your tests. Tests are located inside of functions whose names begin with "test". Those functions are located inside of classes that subclass Django's TestCase. Those classes live in files whose names also begin with "test". To run your tests, type '`python manage.py test`'. More information about testing in Django can be found here: https://docs.djangoproject.com/en/4.0/topics/testing/overview/
- `runserver`: This runs the system on a localhost server on your computer. To run the server, '`python manage.py runserver`', then open your web browser and navigate to http://127.0.0.1:8000/

Coverage.py

Coverage.py (https://coverage.readthedocs.io/en/6.3.2/) is a third-party tool installed in our system for code coverage reports. To calculate the code coverage, run the command "`coverage run manage.py test`" on the terminal. Then, to create a command-line report of the code coverage, run "`coverage report`". To create an HTML report of the code coverage, run "`coverage html`" and then navigate to the index.html file in the `htmlcov` directory. You can specify additional instructions for the code coverage tool, such as a list of files and directories to ignore when calculating code coverage, in the `.coveragerc` file.

Cloc

cloc (https://github.com/AlDanial/cloc) is a third-party tool installed in our system for counting lines of code. You can calculate the lines of code by simply by typing '`cloc`' in the terminal, though it may count lines of code in files you'd prefer it didn't. The file `.clocignore` specifies files and directories for cloc to ignore when counting lines of code. For a more accurate count, we recommend running the command "`cloc --exclude-list-file=.clocignore --exclude-ext=svg .`"

Documentation

Documentation for the system (including this document) can be found in the `docs` directory.

Summary

That about concludes our summary of the ProofChecker system. We hope that after reading this document you feel prepared to maintain and extend the system. If you are still feeling stuck, feel free to reach out to Colton Shoenberger at cmshoenberger@gmail.com with any questions you may have.