University of Nebraska - Computer Science Tutoring Portal

Release 0.0.1

Nolan Gregory

Dec 11, 2023

CSLC PORTAL BACKEND

CHAPTER

ONE

ABOUT

CHAPTER

TWO

API ENDPOINTS

2.1 Course

class api.endpoints.course.**APICourseList**(***kwargs*)

Bases: APIView

get(*request: Request*) → Response

get_querystring(*request: Request*) → QueryDict | Any

post(*request: Request*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

sanitize(*querystring: str*) → str

serializer_class

alias of CourseSerializer

2.2 Issues

class api.endpoints.issue.**APIIssueDetail**(***kwargs*)

Bases: APIView

get(*request: Request*, *pk: str | Any = Ellipsis*) → Response

put(*request: Request*, *pk: str | Any = Ellipsis*) → Response

query_obj(*pk: str*) → QuerySet | Any

class api.endpoints.issue.**APIIssueView**(***kwargs*)

Bases: APIView

get(*request: Request*) → Response

post(*request: Request*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

serializer_class

alias of IssueSerializer

2.3 Professor

class api.endpoints.professor.**APIProfessorDetail**(***kwargs*)

Bases: APIView

get(*request: Request*, *professor_pk: str*) → Response

put(*request: Request*, *professor_pk: str*) → Response

query_obj(*pk: str*) → QuerySet

Add more fish or shrimp to the tank.

Parameters

- **inhabitant** – The type of inhabitant, either shrimp of fish
- **quantity** – The number of fish or shrimp to be added

Raises

TankIsFullError – if the tank is already full

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

serializer_class

alias of ProfessorSerializer

class api.endpoints.professor.**APIProfessorView**(***kwargs*)

Bases: APIView

get(*request: Request*) → Response

get_querystring(*request: Request*) → QueryDict | Any

post(*request: Request*, *search: str | None = None*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

sanitize(*querystring: str*) → str

serializer_class

alias of ProfessorSerializer

2.4 Section

class api.endpoints.section.**APISectionDetail**(***kwargs*)

Bases: APIView

get(*request: Request*, *section_id: str*) → Response

put(*request: Request*, *section_id: str*) → Response

query_obj(*pk: str*) → QuerySet | None

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

serializer_class

alias of SectionSerializer

class api.endpoints.section.**APISectionView**(***kwargs*)

Bases: APIView

get(*request: Request*) → Response

get_querystring(*request: Request*) → QueryDict | Any

post(*request: Request*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

sanitize(*querystring: str*) → str

serializer_class

alias of SectionSerializer

2.5 Ticket

class api.endpoints.ticket.**APITicketDetail**(***kwargs*)

Bases: APIView

get(*request: Request*, *ticket_id: int*) → Response

patch(*request: Request*, *ticket_id: int*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

serializer_class

alias of TicketGetSerializer

class api.endpoints.ticket.**APITicketView**(***kwargs*)

Bases: APIView

get(*request: Request*) → Response

get_querystring(*request: Request*) → Any

post(*request: Request*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

sanitize(*querystring: str*) → str

serializer_class

alias of TicketSerializer

2.6 User

class api.endpoints.user.**APIUserDetail**(***kwargs*)

Bases: APIView

get(*request: Request*, *user_id: str*) → Response

put(*request: Request*, *user_id: str*) → Response

query_obj(*pk: str*) → QueryDict | Any

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

serializer_class

alias of UserSerializer

class api.endpoints.user.**APIUserView**(***kwargs*)

Bases: APIView

get(*request: Request*) → Response

get_querystring(*request: Request*) → QueryDict | Any

post(*request: Request*) → Response

renderer_classes = (<class 'rest_framework.renderers.BrowsableAPIRenderer'>, <class 'rest_framework.renderers.JSONRenderer'>, <class 'rest_framework.renderers.HTMLFormRenderer'>)

sanitize(*querystring: str*) → str

serializer_class

alias of UserSerializer

CHAPTER

THREE

DATABASE MODELS

3.1 Course

class api.models.course.**Course**(**args*, ***kwargs*)

Bases: Model

The course table holds all information about a specific course.

Important: A course can have many sections, while a section can only be attributed to a single course. The course represents what you would see in a catalog (e.g., CIST-1400 & CSCI-1620), and sections are specific instances of the course (online, in person, etc.). Unlike sections, a course does not have a directly assigned professor; this data is stored in sections.

---

course_department

The department code of the course (e.g., "CSCI").

Type
CharField

course_name

The name of the course as it appears

Type
CharField

in the catalog

Type
e.g., "Operating Systems"

course_id

The unique identifier for the course.

Type
IntegerField

course_code

The code associated with the course (e.g., "4500").

Type
CharField

Manager:
generic (Manager): The default manager for the *Course* model.

Example

Examples can be given using either the `Example` or `Examples` sections. Sections support any reStructuredText formatting, including literal blocks:

```
$ python example_google.py
```

Todo:

- For module TODOs
- You have to also use `sphinx.ext.todo` extension

---

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

course_code

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

course_department

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

course_id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

course_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

generic: Manager = <django.db.models.manager.Manager object>

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

section_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

ticket_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

usertocoursetaken

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

usertocoursetutored

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

3.2 Issues

class api.models.issue.**IssueManager**(**args*, ***kwargs*)

Bases: `Manager`

get_issues() → QuerySet

class api.models.issue.**Issues**(**args*, ***kwargs*)

Bases: `Model`

Issues are a field in the ticket that a user will choose to describe the issue they are having. This allows for ticket issues to be comprehensively studied, and provides cleaner data as opposed to users entering their issue type manually. With this style of ticket issue, data analysis can be done more rapidly and with a higher confidence than previous methods allowed. The fields in the table are the problem type (i.e. homework), the severity (used for data collection purposes). The admin is able to quickly add and modify the issues with no consequence to the attributed tickets.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

generic: Manager = <django.db.models.manager.Manager object>

get_severity_display(*, *field=<django.db.models.fields.CharField: severity>*)

issue_id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

problem_type

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

severity

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

ticket_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

3.3 Professor

class api.models.professor.**Professor**(**args*, ***kwargs*)

Bases: `Model`

The professor table holds all details attributed to a professor. This table contains a professors first name, their last name, their full name, their email address, whether they are currently active (see "semester" for more detail on that), and their prof. ID. Since the professor ID is guranteed to be unique, we have opted to use this as the primary key for the table. As such, professors are searchable by their unique ID, and the serializer will likewise return the str representation as well.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

email

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

first_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

full_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

generic = <django.db.models.manager.Manager object>

is_active

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

last_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

professor = <api.models.professor.ProfessorManager object>

professor_id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

section_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

ticket_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class api.models.professor.**ProfessorManager**(**args*, ***kwargs*)

Bases: `Manager`

get_professor(*professor: str*) → QuerySet

get_professors() → QuerySet

3.4 Section

3.5 Ticket

class api.models.ticket.**Ticket**(**args*, ***kwargs*)

Bases: Model

The base class for tickets in the database. This is where the meat of the application purpose lies, as this table will hold all fields associated with a specific ticket. These fields are the professor, the section, the semester, the issue, the student who submitted the ticket, the tutor who primarily helped with the ticket, the tutor(s) who assisted the primary tutor, the name of the student, whether it was a successful ticket, the time the ticket was created, the date the ticket was created, the time the ticket was claimed (different than created), the time the ticket was closed, additional tutor comments, and if the ticket was reopened after it was closed.

CLOSED = 'CLOSED'

DIFFICULTY_CHOICES = [('EASY', 'Easy'), ('MEDIUM', 'Medium'), ('HARD', 'Hard')]

exception DoesNotExist

Bases: ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

NEW = 'NEW'

OPENED = 'OPENED'

STATUS_CHOICES = [('NEW', 'New'), ('OPENED', 'Opened'), ('CLOSED', 'Closed')]

asst_tutor

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Child.parent is a ForwardManyToOneDescriptor instance.

asst_tutor_id

closed_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

course

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Child.parent is a ForwardManyToOneDescriptor instance.

course_id

created_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

description

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

difficulty

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

generic = <django.db.models.manager.Manager object>

get_difficulty_display(*, *field=<django.db.models.fields.CharField: difficulty>*)

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

get_status_display(*, *field=<django.db.models.fields.CharField: status>*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

issue

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

issue_id

messages_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

opened_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

professor

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

professor_id

status

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

student

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

student_id

ticket = <api.models.ticket.TicketManager object>

title

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

tutor

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

tutor_id

updated_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

was_flagged

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

was_reopened

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

was_successful

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class api.models.ticket.**TicketManager**(**args*, ***kwargs*)

Bases: Manager

get_active() → QuerySet

get_all() → QuerySet

get_completed() → QuerySet

get_course(*course: str*) → QuerySet

get_professor(*professor: str*) → QuerySet

get_section(*section: str*) → QuerySet

get_student(*student: str*) → QuerySet

get_successful() → QuerySet

get_tutor(*tutor: str*) → QuerySet

get_unclaimed() → QuerySet

3.6 User

class api.models.user.**AdminManager**(**args*, ***kwargs*)

Bases: Manager

get_admins() → QuerySet

class api.models.user.**StudentManager**(**args*, ***kwargs*)

Bases: Manager

get_student(*name: str*) → QuerySet

get_students() → QuerySet

class api.models.user.**TutorManager**(**args*, ***kwargs*)

Bases: Manager

get_tutor(*name: str*) → QuerySet

get_tutors() → QuerySet

class api.models.user.**User**(**args*, ***kwargs*)

Bases: Model

Generic user model for all active users of the application. Flags are used to indicate whether a user is an admin, a tutor, or a regular student. This currently will hold all data, such as tutor and admin specific data, but this might lead to empty fields within many of the objects in the table. This will be updated in the future to allow specific roles to be relations in the database, thus preventing empty fields (i.e. empty cells).

exception DoesNotExist

Bases: ObjectDoesNotExist

MSOID

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

admin = <api.models.user.AdminManager object>

courses_taken

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

Pizza.toppings and Topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

courses_tutoring

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

Pizza.toppings and Topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

email

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

generic = <django.db.models.manager.Manager object>

hour_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

is_active

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

is_admin

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

is_tutor

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

is_working

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

messages_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

student = <api.models.user.StudentManager object>

student_nuid

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

student_ticket

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

tutor = <api.models.user.TutorManager object>

tutor_ticket

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

user_bio

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

workinghours_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

CHAPTER

FOUR

API SCRIPTS

CHAPTER

FIVE

API CONFIG

5.1 Settings

5.2 base.asgi module

ASGI config for University-Nebraska-Tutor-Portal project.

It exposes the ASGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/4.2/howto/deployment/asgi/

5.3 base.settings module

Settings for University-Nebraska-Tutor-Portal project.

5.4 base.urls module

URL configuration for University-Nebraska-Tutor-Portal project.

The *urlpatterns* list routes URLs to views. For more information please see:
https://docs.djangoproject.com/en/4.2/topics/http/urls/

Examples: Function views

1. Add an import: from my_app import views
2. Add a URL to urlpatterns: path(‘’, views.home, name=’home’)

Class-based views

1. Add an import: from other_app.views import Home
2. Add a URL to urlpatterns: path(‘’, Home.as_view(), name=’home’)

Including another URLconf

1. Import the include() function: from django.urls import include, path
2. Add a URL to urlpatterns: path(‘blog/’, include(‘blog.urls’))

5.5 base.wsgi module

WSGI config for University-Nebraska-Tutor-Portal project.

It exposes the WSGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/4.2/howto/deployment/wsgi/

5.6 Module contents

CHAPTER

SIX

FORMS

6.1 TicketForm TypeScript Module Documentation

The *TicketForm* TypeScript module defines a React component responsible for creating support tickets. This documentation provides an overview of the module's structure, key components, and their responsibilities.

6.1.1 Overview

The *TicketForm* component facilitates the creation of support tickets by gathering information from users, such as their name, selected professor, course, issue type, and a detailed description of the problem. It leverages various utility functions, hooks, and external APIs to enhance its functionality.

6.1.2 Key Components

FormSchema

The *FormSchema* is a Zod schema defining the shape and validation rules for the form data. It ensures that the user provides valid input for fields such as name, title, professor, course, issue, and description.

TicketLabel Component

This component renders a formatted label for certain form fields, such as the full name.

TicketDescription Component

Similar to *TicketLabel*, this component renders a description for specific form fields, providing additional guidance or context.

TicketForm Component

The main *TicketForm* component orchestrates the entire form. It uses the *react-hook-form* library for form management and validation. The component includes form fields for user information, selects for professors, courses, and issue types, and a text area for the ticket description. It also handles form submission, triggering the creation of a support ticket through the *createTicket* API.

useFetchProfessor, useFetchCourse, useFetchIssue Hooks

These custom hooks fetch data from external APIs related to professors, courses, and issue types, respectively. They ensure that the form has up-to-date information for dropdowns and selects.

LoadingSelect Component

This component provides a visual indication (e.g., loading spinner) when fetching data for selects.

useToast Hook

The *useToast* hook manages the display of toast messages, providing user feedback on form submission.

useNavigate Hook

The *useNavigate* hook from *react-router-dom* facilitates navigation upon successful ticket submission.

6.2 TicketForm React Component Documentation

The *TicketForm* component is a React form used to create a new ticket in a ticketing system. It allows users to provide information about their name, the professor, course, issue type, a brief summary, and a detailed description of their ticket. The form incorporates various UI components, including buttons, inputs, and popovers.

6.2.1 Overview

The *TicketForm* component comprises several key sections and components:

- **Form Initialization**: The form is initialized using the *useForm* hook from *react-hook-form*. It uses the *FormSchema* provided by the *zod* library for validation.
- **Form Submission**: The form submission is handled by the *onSubmit* function, which triggers the *createTicket* mutation and displays a success toast message.
- **LoadingSelect**: A loading state is displayed for select dropdowns while fetching data.
- **Popovers and Commands**: Popovers are used for professor, course, and issue type selection, incorporating a search functionality (*Command*). Each selection dropdown is triggered by a *Button* and displays a list of items with a search input.
- **Form Fields and Validation**: Various form fields are utilized, including *Input*, *Textarea*, and custom *FormField* components. The form includes validation for name, title, professor, course, issue type, and description.
- **TicketLabel and TicketDescription**: Custom components (*TicketLabel* and *TicketDescription*) are used for styling labels and descriptions consistently.
- **Button Component**: The form submission is triggered by a *Button* component with dynamic disabling based on the mutation's pending state.

6.2.2 Usage

To use the *TicketForm* component, integrate it into the desired page or component. The form handles the creation of new tickets, including input validation and mutation handling.

Example:

```jsx import TicketForm from "@/path/to/TicketForm";

**function NewTicketPage() {**

    **return (**

        **<div>**
            <h1>Create a New Ticket</h1> <TicketForm />

        </div>

    );

}
```

Additional Notes

- The *max_ticket_length* constant defines the maximum character limit for the ticket description.
- The form utilizes the *zodResolver* from *@hookform/resolvers/zod* for Zod schema-based validation.
- The *useMutation* hook from *@tanstack/react-query* manages the asynchronous ticket creation process.

6.2.3 Usage

To use the *TicketForm* component, integrate it into a parent component or page within a React application. Make sure to include the necessary dependencies and handle form submission accordingly.

Example:

```jsx import TicketForm from "@/path/to/TicketForm";

**function SupportPage() {**

    **return (**

        **<div>**
            <h1>Submit a Support Ticket</h1> <TicketForm />

        </div>

    );

}
```

6.3 AnnouncementForm

This module contains the *AnnouncementForm* component, which is used to create and submit announcements.

6.3.1 Form Schema

The *FormSchema* object defines the schema for the announcement form, including the title, body, variant, start date, end date, and show date. It also specifies the validation rules and error messages for each field.

6.3.2 Submitting the Form

When the form is submitted, the *onSubmit* function is called, which triggers a toast notification displaying the submitted values in a formatted JSON style.

6.3.3 Rendering the Form

The *AnnouncementForm* component renders a form using the *useForm* hook from *react-hook-form*. It includes input fields for the announcement title, body, variant, start date, end date, and show date. Each field is rendered with appropriate validation and error messages.

6.3.4 UI Components

The form utilizes various UI components such as *Textarea*, *Select*, *Input*, *Button*, *Switch*, *Popover*, and *CalendarIcon* for user interaction and input.

6.3.5 Character Limits

The form also enforces character limits for the body field, with a maximum length of 255 characters. The character count is dynamically displayed as the user types in the *Textarea*, providing real-time feedback to the user.

6.3.6 Form Submission

Upon completion, the user can submit the form by clicking the "Schedule" button.

This RST documentation provides an overview of the *AnnouncementForm* component and its functionality within the ticketing portal.

6.4 ThemeProvider React Component Documentation

The *ThemeProvider* component is a React context provider that manages the theme state for a ticketing portal. It allows users to customize the theme (dark, light, or system default) and persists the selected theme in local storage.

6.4.1 Overview

The *ThemeProvider* component is responsible for providing a theme context to its children components. It offers functionality to set and retrieve the current theme, and it automatically syncs the theme with the user's system preference when the "system" theme is selected.

6.4.2 Key Components

createContext, useContext

These functions are used from the *react* library to create and access the theme context, respectively.

useState

The *useState* hook is used to manage the local state of the selected theme.

useEffect

The *useEffect* hook is used to update the document's root element with the selected theme class. It also listens for changes in the theme and adjusts the document accordingly.

ThemeProviderProps

This type defines the props expected by the *ThemeProvider* component, including *children*, *defaultTheme*, and *storageKey*.

ThemeProviderState

This type represents the state of the *ThemeProvider* component, including the current theme and the *setTheme* function.

initialState

This constant represents the initial state of the *ThemeProvider* component, with the default theme set to "system."

ThemeProviderContext

The context created using *createContext* to provide theme-related values to child components.

ThemeProvider Component

The main *ThemeProvider* component that manages the theme state, sets the theme class on the document root, and provides the theme context to its children.

useTheme Hook

The *useTheme* hook is a custom hook that retrieves the current theme and *setTheme* function from the context. It throws an error if used outside a *ThemeProvider*.

6.4.3 Usage

To use the *ThemeProvider* component, wrap it around the components that need access to the theme context within a React application. Customize the theme by providing the desired values for *defaultTheme* and *storageKey*.

Example:

```jsx import ThemeProvider, { useTheme } from "@/path/to/ThemeProvider";

**function App() {**

    **return (**

        **<ThemeProvider defaultTheme="dark">**
            <MainContent />

        </ThemeProvider>

    );

}

**function MainContent() {**
    const { theme, setTheme } = useTheme();

    // Implement theme-specific rendering or functionality based on the theme context.

    **return (**

        **<div>**
            <h1>Theme is {theme}</h1> <button onClick={() => setTheme("light")}>Switch to Light Theme</button>

        </div>

    );

}
```

6.5 ModeToggle React Component Documentation

The *ModeToggle* component is a React component that provides a mode toggle button for changing the theme of a ticketing portal. It utilizes icons for the sun and moon to represent light and dark themes, respectively. The component is implemented using the *lucide-react* library for icon components and leverages the *DropdownMenu* component for the theme selection options.

6.5.1 Overview

The *ModeToggle* component offers a user-friendly way to switch between light, dark, and system (default) themes in the ticketing portal. Users can click on the button to reveal a dropdown menu with theme options.

6.5.2 Key Components

Button Component

The *Button* component from *@/components/ui/button* is used to create the clickable button for toggling themes. It includes icons for the sun and moon, providing a visual representation of the available themes.

DropdownMenu Components

The *DropdownMenu*, *DropdownMenuTrigger*, *DropdownMenuContent*, and *DropdownMenuItem* components from *@/components/ui/dropdown-menu* are utilized to create a dropdown menu for selecting different themes. The menu is triggered by clicking on the mode toggle button.

useTheme Hook

The *useTheme* hook from *@/forms/ThemeProvider* is used to access the *setTheme* function, enabling the component to update the selected theme based on user input.

6.5.3 Usage

To use the *ModeToggle* component, integrate it into a parent component or page within a React application. Ensure that the necessary dependencies are imported, and handle theme changes accordingly.

Example:

```jsx import ModeToggle from "@/path/to/ModeToggle";

**function ThemeSettingsPage() {**

    **return (**

        **<div>**
            <h1>Theme Settings</h1> <ModeToggle />

        </div>

    );

}
```

6.6 TutorTicketForm React Component Documentation

The *TutorTicketForm* component is a React form used in a ticketing portal to display and update ticket details. It includes various form fields for ticket information and provides functionality to update the ticket using the *updateTicket* API. The form is built using the *react-hook-form* library and incorporates components from the project's UI library.

6.6.1 Overview

The *TutorTicketForm* component consists of several key sections and components:

- **Alert Dialog and Trigger**: The form is displayed within an *AlertDialog*, triggered by a button with three dots.
- **Form Initialization**: The form is initialized using the *useForm* hook from *react-hook-form*. It uses a *FormSchema* provided by the *zod* library for validation.
- **Form Submission**: The form submission is handled by the *onSubmit* function, which triggers the *updateTicket* mutation and displays a success toast message.
- **Ticket Details Display**: Ticket details, including the title, ID, status, and actions, are displayed at the top of the form.
- **Dropdown Menus**: The form includes two dropdown menus with various actions, such as copying ticket details or flagging the ticket.
- **CheckDropField**: A custom checkbox field (*CheckDropField*) is used to toggle the *was_successful* property.
- **TextareaField**: A custom textarea field (*TextareaField*) is used for entering and displaying the ticket description.
- **DropdownField**: A custom dropdown field (*DropdownField*) is used for selecting the ticket status.
- **SearchFilterField and DropField**: Custom fields are used for selecting tutors and difficulty levels, respectively.
- **DetailLink Component**: The *DetailLink* component displays labeled details, such as student, professor, and course information.
- **Button Components**: Various buttons are used for actions like updating the form, discarding changes, or copying ticket details.

6.6.2 Usage

To use the *TutorTicketForm* component, pass the ticket object as a prop. The form handles the display and editing of ticket details. It relies on the *react-hook-form* library for form management and incorporates components from the project's UI library for consistent styling.

Example:

```jsx
import TutorTicketForm from "@/path/to/TutorTicketForm";

function TicketDetailsPage({ ticket }) {
    return (
        <div>
            <h1>Ticket Details</h1> <TutorTicketForm ticket={ticket} />
        </div>
    );
}
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

b

INDEX

G

H

I

L

M

N

O

P

Q

R

S

T

U

W