

---

# BUILDING A KIVY APPLICATION USING DJANGO

# django



# kivy

In this article, we are going to learn how to create a [Kivy](#) application, and how to make requests to a server using the requests library.

---

Kivy is a popular library for developing media-rich, multi-touch enabled applications that can run across all major platforms. However, there may be situations that need your application to get access to and manipulate data provided by a web server hosted at a site.

In this article, we are going to learn how to create a Kivy application, and how to make requests to a server using the requests library.

We are going to use Django to develop the server that holds the data that we want to manipulate.

# PREREQUISITES

To follow along, it's important that:

- You have Django and `djangorestframework` frameworks installed.
- You are familiar with building APIs using the Django REST framework.
- You have the `kivy` library installed. If you have not installed it, you can run `pip install kivy`.
- Basic knowledge of Python is important. Knowing Object-Oriented Programming would help.

## KEY TAKEAWAYS

- Improve your python skills.
- Learn how to build applications using the kivy library.
- Learn how to use Django for your kivy applications.

# GETTING STARTED

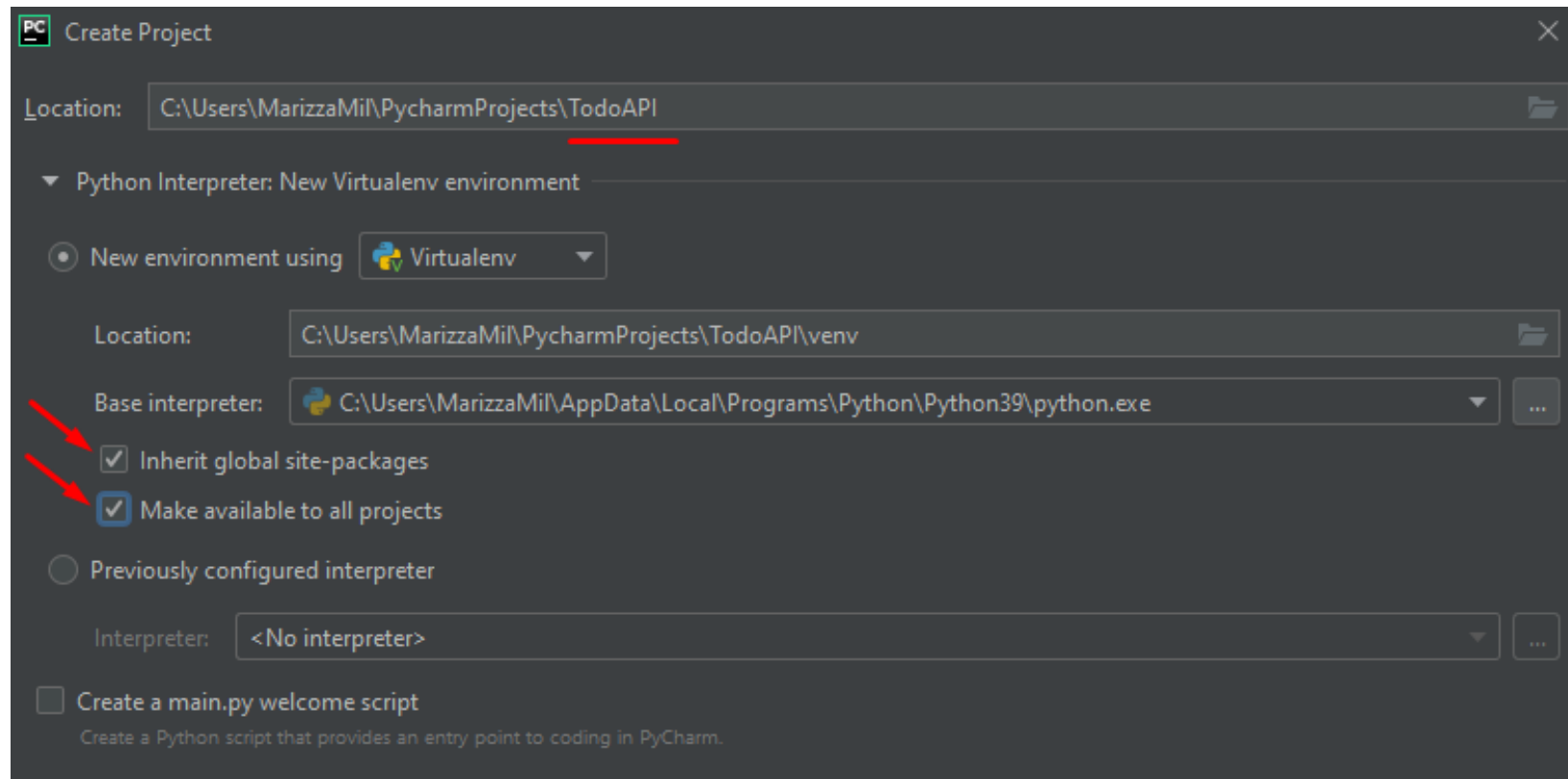
We are going to create a simple to-do application that allows one to view available tasks, offers an option for adding a new task.

We will begin by creating the tasks API with `django rest framework` and then create our application using `kivy`.

We will use the `requests` library to make requests to our Django server.

---

# CREATE NEW PROJECT IN PYCHARM



---

# CREATING THE TASKS API

- Install Django, DjangoREST and Kivi:

```
$ pip install django django-rest-framework django-cors-headers
```

- Create a new project by running:

```
$ django-admin startproject TodoAPI .
```

- In the TodoAPI project, create a new app named `tasks` that will handle the creation of tasks.

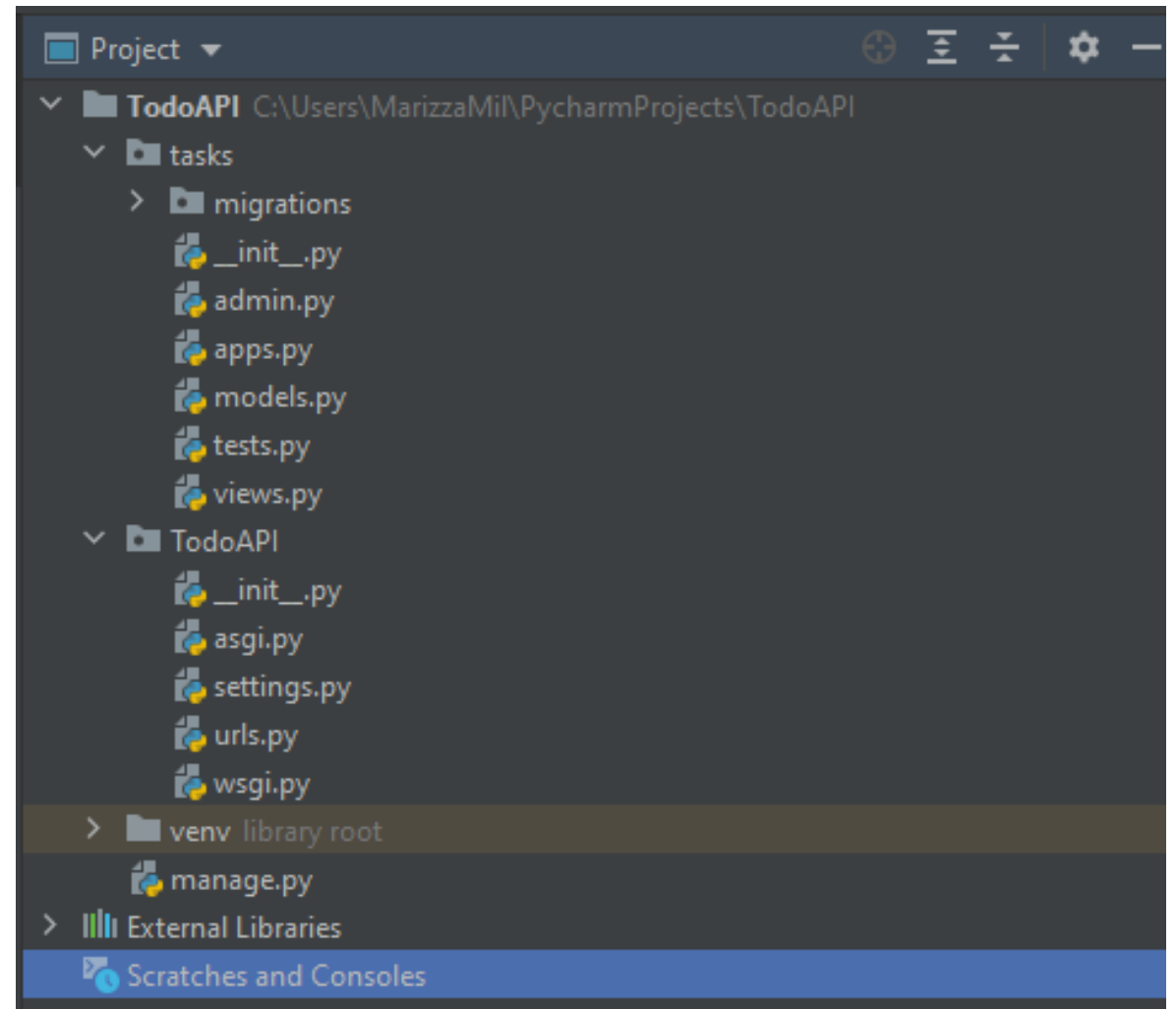
Run the command below to create the app:

```
$ python manage.py startapp tasks
```

---

---

YOUR PROJECT  
STRUCTURE SHOULD  
LOOK SOMETHING LIKE  
THIS





---

# CREATING THE TASKS API

We need to register our tasks app and `rest_framework` to use the app.

Edit the `settings.py` file as follows under `INSTALLED_APPS`:

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'tasks',
    'rest_framework'
]
```

---

# CREATING THE TASKS API

Let's go ahead and create our tasks model. Edit the `tasks/models.py` file to look as follows:

```
from django.db import models

class Task(models.Model):
    name = models.CharField(max_length=200)

    def __str__(self):
        return self.name
```

---

# CREATING THE TASKS API

Let's create a `serializer.py` file that handles the serialization and de-serialization of task instances.

Serialization is the process of saving an object in a way that can be accessed in a uniform format by different applications.

De-serialization is the reverse of serialization.

You may learn more about serialization and de-serialization in the Django REST framework from [this site](#).

Add the following code to the `tasks/serializers.py` file:

```
from rest_framework import serializers
from .models import *

class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = '__all__'
```

---

# CREATING VIEWS

We are going to create some views to render data to a web page.

Views are python functions that handle web requests and return web responses. There are different ways to create the views, we will use function-based views.

Function-based views are views in Django written as python functions.

Edit the `tasks/views.py` file to match the following:

```
from django.shortcuts import render
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response

from .serializers import *
from .models import *

# Create your views here.
@api_view(['GET'])
def all_tasks(request):
    tasks = Task.objects.all()
    serializer = TaskSerializer(tasks, many=True)
    return Response(serializer.data, status=status.HTTP_200_OK)

@api_view(['POST'])
def create_task(request):
    data = request.data
    serializer = TaskSerializer(data=data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_200_OK)
```

---

# ROUTING VIEWS

We then create a `tasks/urls.py` file for routing our views.

Create the file and add the following code:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.all_tasks, name='all_tasks'),
    path('create', views.create_task, name='create_task')
]
```

The mappings above imply that the requests will first be handled by this file and then routed to a corresponding view function.

For instance, when we visit `http://127.0.0.1:8000/create`, the `create_task` function is called and implemented.

We have to create a route for our tasks app.

This way, the routing first occurs on the `TodoAPI/urls.py` file, and then `tasks/urls.py`.

Therefore, configure our `urls.py` file in `TodoAPI` like this:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('tasks.urls'))
]
```

---

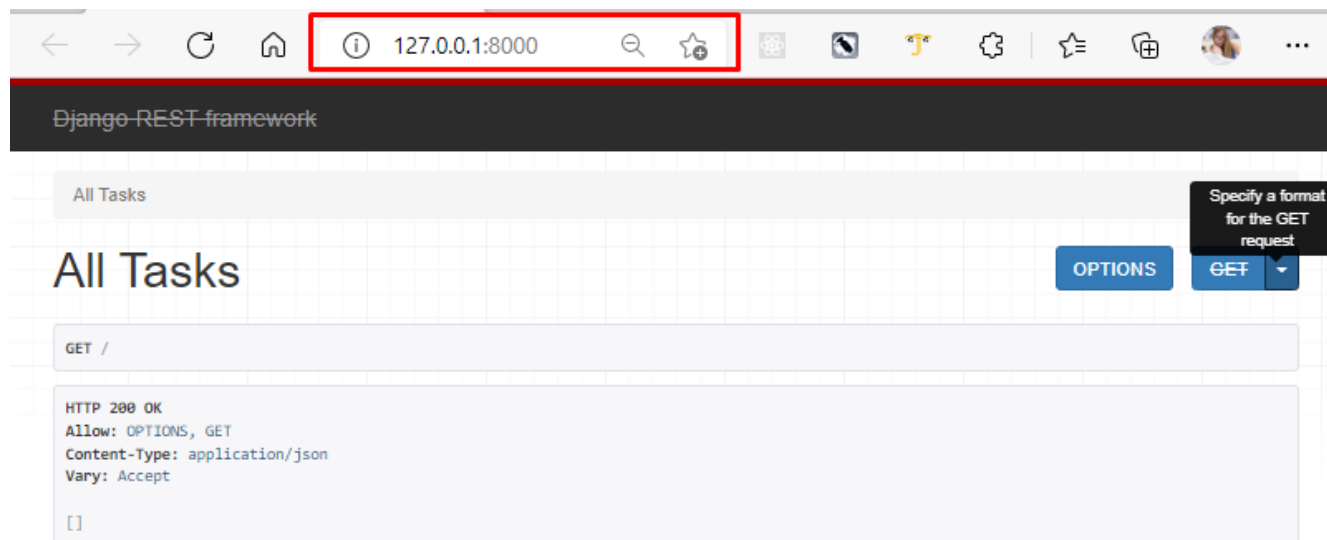
We can now run:

```
$ python manage.py makemigrations
```

```
$ python manage.py migrate
```

```
$ python manage.py runserver
```

We should have a page similar to this:



When you create a task instance as follows:

"

Our Django API is working! We can now proceed to create the `kivy` application.

## CREATING THE KIVY APPLICATION

Let's create a main.py file in a folder of choice.

Here, we'll be using the same directory as our TodoAPI project folder.

We are going to get started with these lines of code in the main.py file.

The TodoApp is the main entry point of our application and every execution begins from there.

[YOUR-FOLDER]/main.py

```
from kivy.app import App

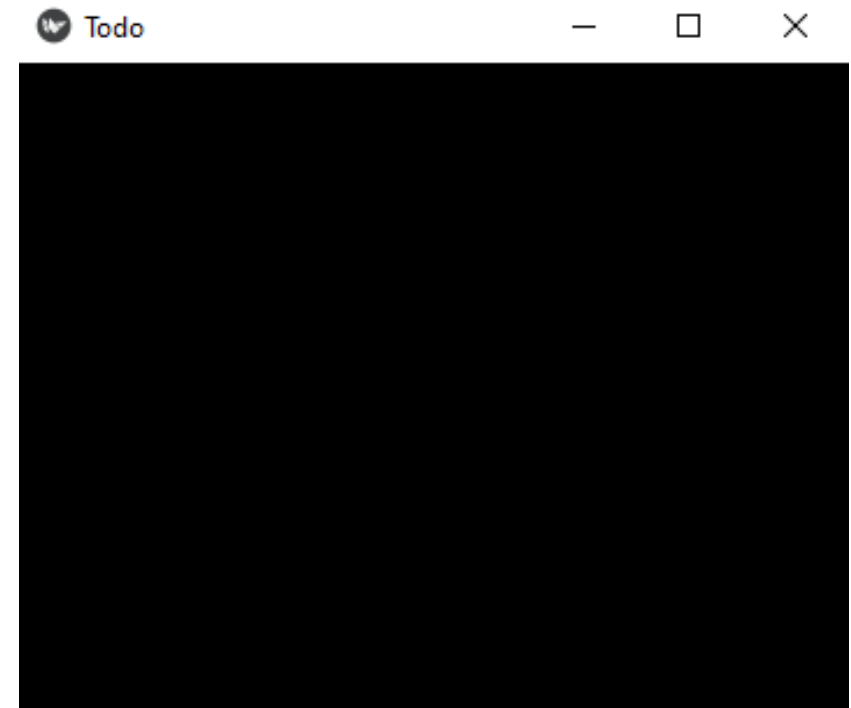
class TodoApp(App):
    pass

if __name__ == '__main__':
    TodoApp().run()
```

When you run the file,

```
$ python main.py runserver
```

you should get the following  
result:



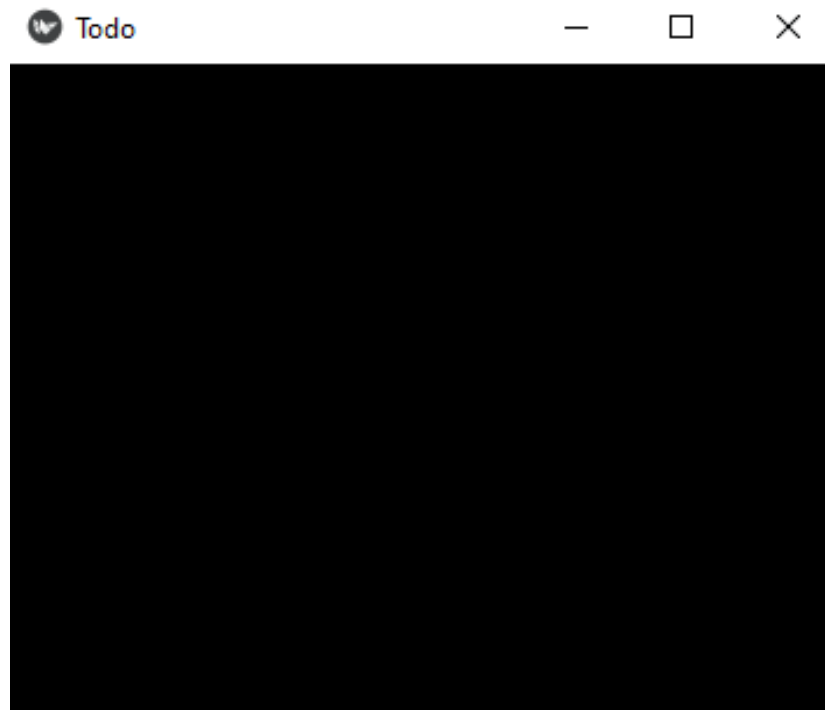


## CREATING THE KIVY APPLICATION

When you run the file,

```
$ python main.py runserver
```

you should get the following result:



# CREATING THE KIVY APPLICATION

We are now going to replace the `main.py` file with the following lines of code:

```
from kivy.app import App

from kivy.uix.screenmanager import ScreenManager
from kivy.uix.boxlayout import BoxLayout

# menu
class Menu(BoxLayout):
    pass

# Screen Management
class ScreenManagement(ScreenManager):
    pass

# app class
class TodoApp(App):
    pass

if __name__ == '__main__':
    TodoApp().run()
```

# CREATING THE KIVY APPLICATION

We also need to create a kv file where all details of user interface is entailed.

Create a todo.kv file and make sure it's in the same folder as the main.py file. The file should resemble this one:

```
BoxLayout:
    orientation: 'vertical'
    Menu:
        size_hint_y: .1
        manager: screen_manager
    ScreenManagement:
        size_hint_y: .9
        id: screen_manager
<Menu>:
    orientation: "vertical"
    ActionBar:
        top: 0
    ActionView:
        ActionPrevious:
        ActionButton:
            text: 'Home'
        ActionButton:
            text: 'Add New'
```

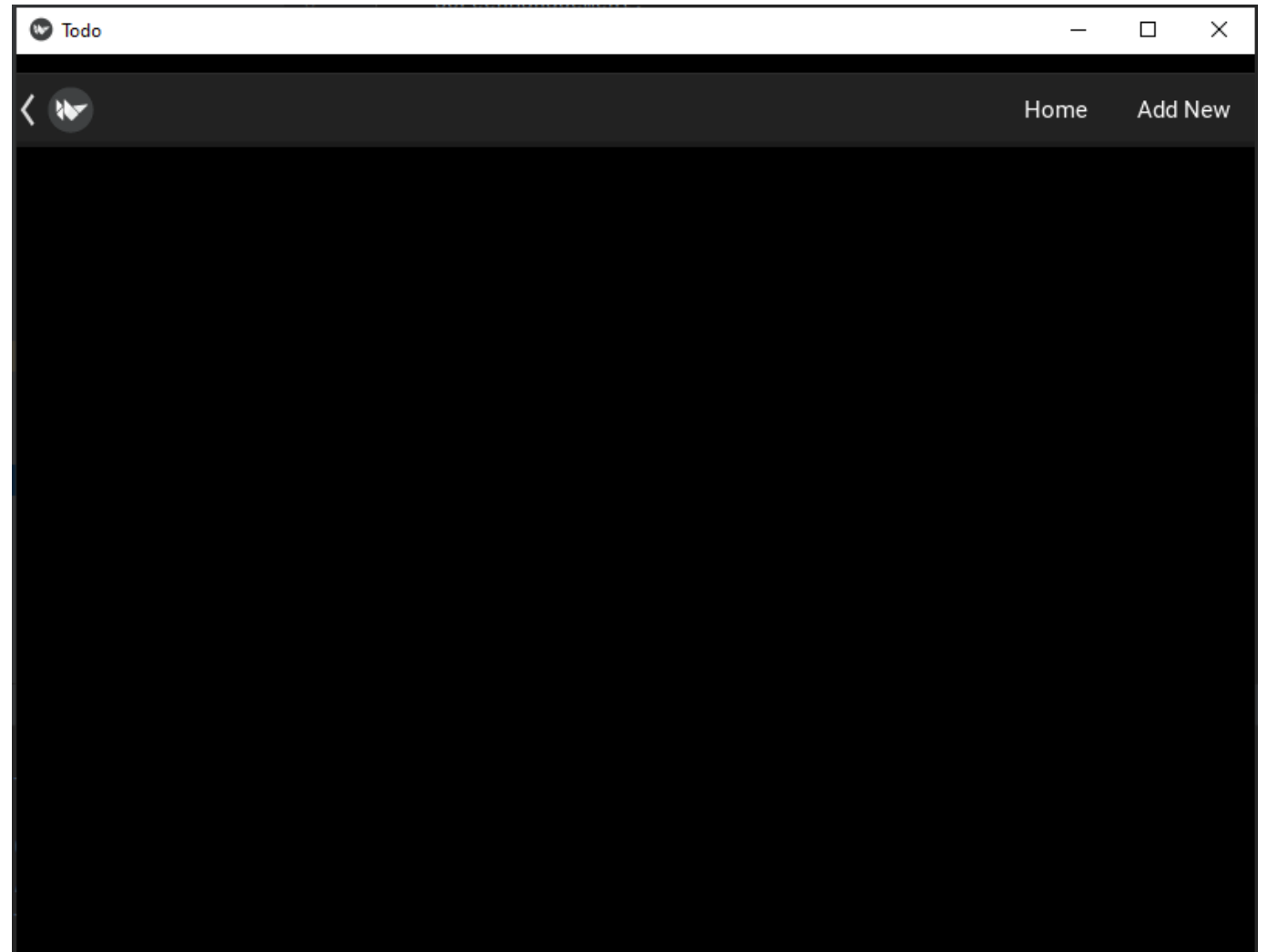
This is what is happening to our files:

- We begin by creating a Menu class that inherits from BoxLayout class. This menu will contain the buttons that we will use to explore the application.
- We then proceed by creating the ScreenManger class that manages the display of what is being displayed in the application.
- In the todo.kv file, we declare a BoxLayout as the main interface that the application displays.
- We then give the menu a position of bottom set by size\_hint\_y: .1.
- We also declare that it's managed by ScreenManager. We do this by declaring manager: screen\_manager.
- By setting the id property of the ScreenManager as screen\_manager, the position of the menu is now on top.
- We declare the properties of our Menu class.
- The class will have an action bar that will contain two buttons, the Home action button, and the Add New action button.

---

# CREATING THE KIVY APPLICATION

Your application should be similar to the one below:



## CREATING THE KIVY APPLICATION

We now need to transition to a different screen when creating a task.

Therefore, we need to declare two screens so that one displays the tasks, and another one to add a new task. Both of these screens will be managed by ScreenManager class.

Edit your main.py file to look like this:

```
from kivy.app import App
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.uix.boxlayout import BoxLayout

# menu
class Menu(BoxLayout):
    pass

# screens
class HomeScreen(Screen):
    pass
class AddScreen(Screen):
    pass

# Screen Management
class ScreenManagement(ScreenManager):
    pass

# app class
class TodoApp(App):
    pass

if __name__ == '__main__':
    TodoApp().run()
```

Add  
code

Also, modify the `todo.kv` file to be as shown below:

```
BoxLayout:
    orientation: 'vertical'
    Menu:
        size_hint_y: .1
        manager: screen_manager
    ScreenManagement:
        size_hint_y: .9
        id: screen_manager
<Menu>:
    orientation: "vertical"
    ActionBar:
        top: 0
        ActionView:
            ActionPrevious:
            ActionButton:
                text: 'Home'
                on_press: root.manager.current = 'screen_home'
            ActionButton:
                text: 'Add New'
                on_press: root.manager.current = 'screen_add'
<ScreenManagement>:
    id: screen_manager
    HomeScreen:
        name: 'screen_home'
        manager: 'screen_manager'
    AddScreen:
        name: 'screen_add'
        manager: 'screen_manager'
<HomeScreen>:
    Label:
        text: "Home"
<AddScreen>:
    Label:
        text: "Add to list..."
```

Add  
code

In the code above:

- We declare the HomeScreen class and give it a Label instance with the text Home.
- We also do the same for the AddScreen with the label text Add to list....
- We have given both screens as children of ScreenManagement.
- We also specified the functionality of action buttons by enabling access to the screen. We do this by defining `on_press: root.manager.current = 'screen_home'` and `on_press: root.manager.current = 'screen_add'`.

You should now be able to explore the two screens and see the “Home” text in the HomeScreen, and “Add to list...” in the AddScreen

## USING THE DJANGO API

Let's create a class that has a function which sends requests to our server for available tasks.

Under the class Menu(BoxLayout): declaration, add the following lines of code:

```
from kivy.app import App
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.uix.boxlayout import BoxLayout

import requests
from kivy.clock import Clock
from kivy.uix.recycleview import RecycleView

# menu
class Menu(BoxLayout):
    Pass

#recycle view for home screen
class MyRecycleView(RecycleView):
    def __init__(self, **kwargs):
        super(MyRecycleView, self).__init__(**kwargs)
        self.load_data()
        Clock.schedule_interval(self.load_data, 1)

    def load_data(self, *args):
        store = requests.get('http://127.0.0.1:8000/').json()

        list_data = []
        for item in store:
            list_data.append({'text': item['name']})

        self.data = list_data

# screens
...
```

main.py

Add  
code

- The MyRecycleView class is initialized by having a function load\_data that makes requests to the server using the requests library.
- The data is appended to a list containing dictionaries of the tasks with the key text. The function returns the list as a data variable.
- The function is called every second by setting a clock interval of 1.

Modify your `todo.kv` file to look like this:

```
BoxLayout:
...
<Menu>:
...
<ScreenManagement>:
    id: screen_manager
    HomeScreen:
        name: 'screen_home'
        manager: 'screen_manager'
    AddScreen:
        name: 'screen_add'
        manager: 'screen_manager'
```

```
<HomeScreen>:
    BoxLayout:
        orientation: "vertical"
    MyRecyclerView:
```

```
<MyRecyclerView>:
    viewclass: 'Label'
    RecyclerView:
        color: 1,1,1,1
        default_size: None, dp(56)
        default_size_hint: 1, None
        size_hint_y: None
        height: self.minimum_height
        orientation: 'vertical'
```

```
<AddScreen>:
    Label:
        text: "Add to list..."
```

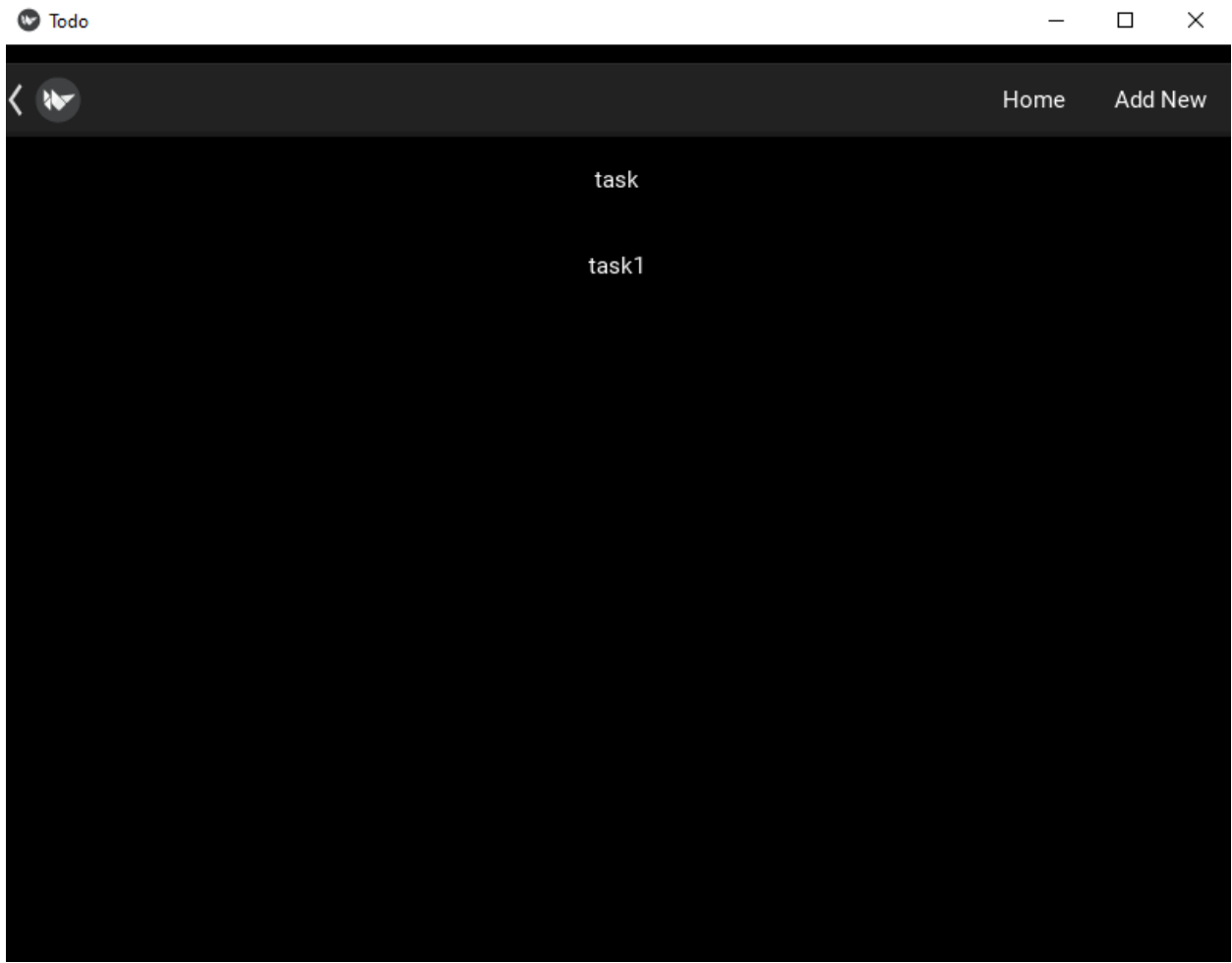
Change  
code

Add  
code

- In the `todo.kv` file, we replace the contents of our `HomeScreen` with a `BoxLayout` that contains our `MyRecyclerView` class.
- We then declare the properties of our `MyRecyclerView` as having `BoxLayout` that contains a list of labels, each with text received from our server.



You should now be able to see the code task we created in our web-based interface earlier.



---

Let us now handle the functionality of creating a new task.

We begin by creating a form to submit the creation request to our server.

We then add the form to our AddScreen screen. This will make our application complete.

Let's modify our `main.py` file to this:

```
from kivy.app import App
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.uix.boxlayout import BoxLayout
import requests
from kivy.clock import Clock
from kivy.uix.recycleview import RecycleView

from kivy.properties import ObjectProperty, StringProperty
from kivy.uix.label import Label
from kivy.uix.widget import Widget

# menu
...
#recycle view for home screen
...
# screens
class HomeScreen(Screen):
    pass
class AddNewForm(Widget):
    text_input = ObjectProperty(None)

    input = StringProperty('')

    def submit_input(self):
        self.input = self.text_input.text
        post = requests.post('http://127.0.0.1:8000/create', json={'name': self.input})

        self.input = ''

class AddScreen(Screen):
    def __init__(self, **kwargs):
        super(AddScreen, self).__init__(**kwargs)
        self.box = BoxLayout()
        self.box.orientation = "vertical"
        self.box.add_widget(Label(text="Add To List...", color="blue", pos_hint={"top": 1}))
        self.addNewForm = AddNewForm()
        self.box.add_widget(self.addNewForm)
        self.add_widget(self.box)

# Screen Management
...
```

Modify your `todo.kv` file to look like this:

```
BoxLayout:
...
<Menu>:
...
<ScreenManagement>:
...
<HomeScreen>:
    BoxLayout:
        orientation: "vertical"
        MyRecyclerView:
<MyRecyclerView>:
    viewclass: 'Label'
    RecyclerView:
        color: 1,1,1,1
        default_size: None, dp(56)
        default_size_hint: 1, None
        size_hint_y: None
        height: self.minimum_height
        orientation: 'vertical'
```

```
<AddNewForm>:
    text_input: input
    TextInput:
        id: input
        pos: root.center_x - 220, 300
        size: 400,50
    Button:
        size: 130,40
        pos: root.center_x - 100, 200
        text: 'Submit'
        on_release: root.submit_input()
```

← Add  
code

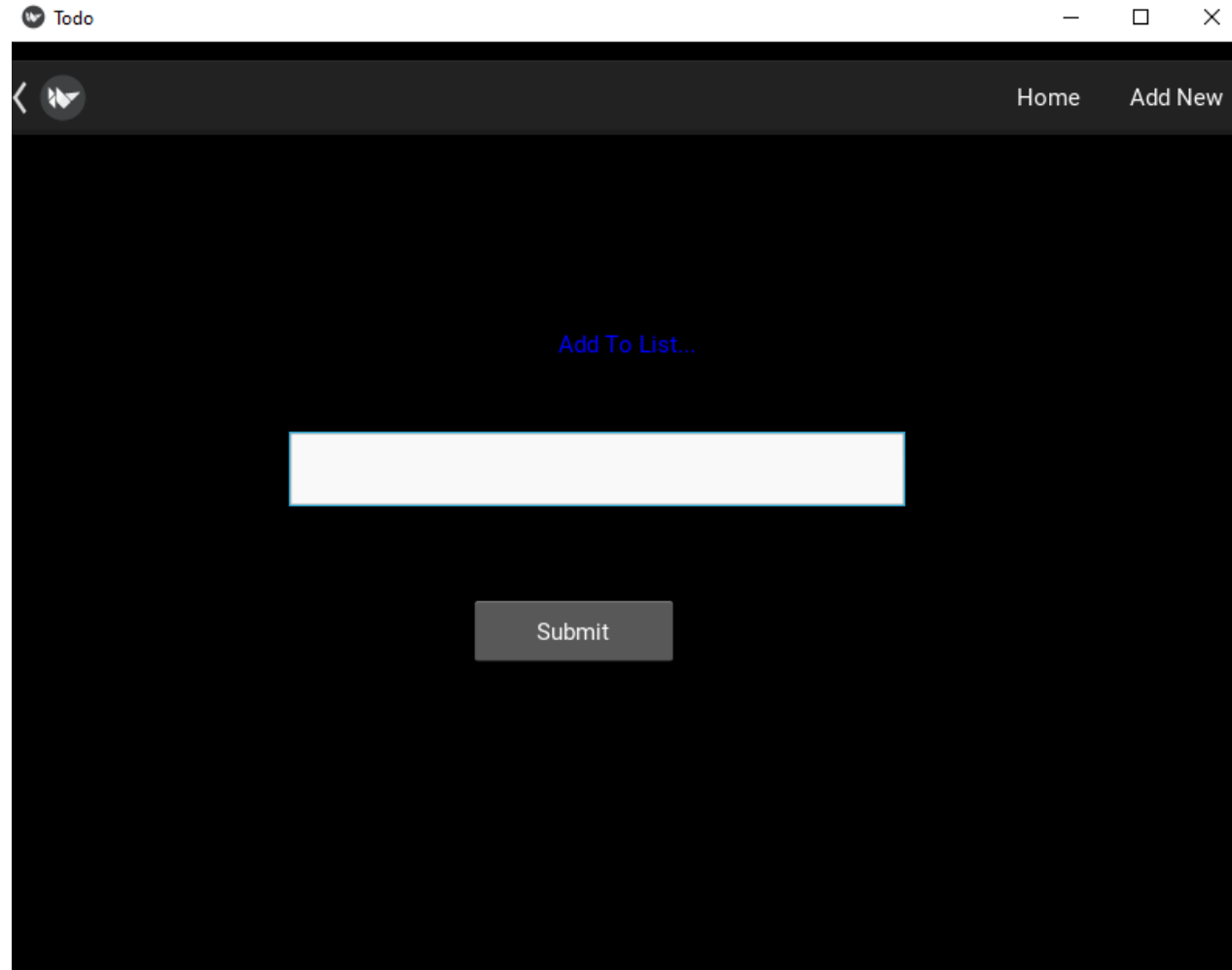
The `AddNewForm` contains a function `submit_input` that makes a POST request to the server passing the input text as data.

The form has a `TextInput` that one can type the task, and a button that calls the `submit_input` function when released.

We then declare a `BoxLayout` class that will contain the `AddNewForm` and a label text “Add to list...”

---

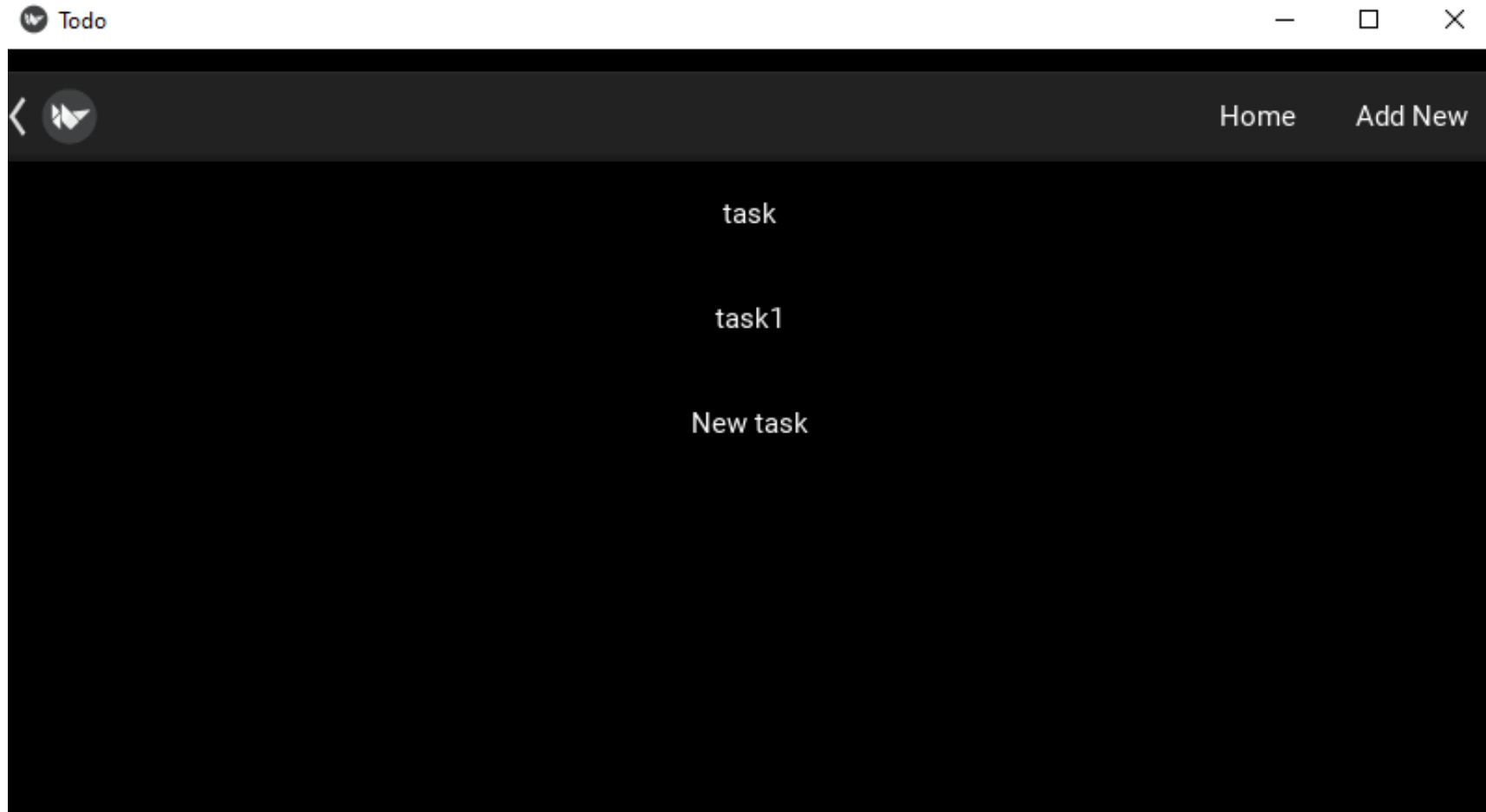
When you now click Add New, you should now see the following:



The screenshot shows a web application window titled "Todo" with standard window controls (minimize, maximize, close). The application has a dark theme. At the top, there is a navigation bar with a back arrow, a home icon, and two links: "Home" and "Add New". The "Add New" link is active. The main content area is dark and contains the text "Add To List..." in blue. Below this text is a white rectangular input field. At the bottom of the input area is a grey button labeled "Submit".

---

When you create a task, say “new task”, and click submit button once then click Home, you should have the following:



# CONCLUSION

---

In this tutorial, we have covered the basics of kivy by creating a simple todo kivy application that allows one to view and add tasks.

We have also seen how we can use Django as a back-end for the application by creating a server that holds our tasks.

With this knowledge, you can create similar applications to suit your different needs.

# FURTHER READING

The following are important links that will help you create kivy applications and build APIs using Django and the Django REST framework:

- [Kivy documentation](#)
- [Django documentation](#)
- [Django REST framework documentation](#)