# WebSockets - Part One

### Part 1, Chapter 6

« HTTP

Up until now, we have dealt with users in a generic way: Users can authenticate and they can retrieve trips. The following section separates users into distinct roles, and this is where things get interesting. Fundamentally, users can participate in trips in one of two ways: they either drive the cars or they ride in them. A rider initiates the trip with a request, which is broadcasted to all available drivers. A driver starts a trip by accepting the request. At this point, the driver heads to the pick-up address. The rider is instantly alerted that a driver has started the trip and other drivers are notified that the trip is no longer up for grabs.

Instantaneous communication between the driver and the rider is vital here, and we can achieve it using WebSockets via Django Channels.

## Connecting to the Server

Clients and servers using the HTTP Protocol establish a single connection *per request*. The client initiates communication and the server responds. It never works the other way around. After the request/response cycle finishes, the connection closes.

On the other hand, clients and servers using the WebSocket protocol establish just one connection *total*. Both the client and the server can send messages to each other over the same open connection until it disconnects.

Let's wade into the topic of WebSockets before diving right in.

Create a new *test_websocket.py* file in the same directory as *test_http.py* and add the following code to it:

```python
# server/trips/tests/test_websocket.py

import pytest
from channels.testing import WebsocketCommunicator

from taxi.asgi import application

TEST_CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels.layers.InMemoryChannelLayer',
    },
}


@pytest.mark.asyncio
class TestWebSocket:
    async def test_can_connect_to_server(self, settings):
        settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
        communicator = WebsocketCommunicator(
            application=application,
            path='/taxi/'
        )
        connected, _ = await communicator.connect()
        assert connected is True
        await communicator.disconnect()
```

Our first WebSocket test proves that a client can connect to the server.

One of the first things you'll probably notice is that we're using `pytest` instead of the built-in Django testing tools. We're also using coroutines that were introduced with the `asyncio` module in Python 3.4. Django Channels mandates the use of both `pytest` and `asyncio`.

> If you're not familiar with asynchronous programming, we'd strongly encourage you to learn the basics, starting with the official asyncio Python documentation and the excellent Python & Async Simplified guide (by the creator of Django Channels, Andrew Godwin). Also, if you're curious about concurrency and parallelism in Python in general, check out the Speeding Up Python with Concurrency, Parallelism, and asyncio blog post.

Remember how we created HTTP test classes by extending `APITestCase`? Grouping multiple tests with pytest only requires you to write a basic class. We've named ours `TestWebsocket`. We've also decorated the class with a mark, which sets metadata on each of the test methods contained within. The `@pytest.mark.asyncio` mark tells `pytest` to treat tests as `asyncio` coroutines.

Pay attention to the fact that we're including a `TEST_CHANNEL_LAYERS` constant at the top of the file after the imports. We're using that constant in the first line of our test along with the `settings` fixture provided by `pytest-django`. This line of code effectively overwrites the application's settings to use the `InMemoryChannelLayer` instead of the configured `RedisChannelLayer`. Doing this allows us to *focus our tests on the behavior we are programming rather than the implementation* with Redis. Rest assured that when we run our server in a non-testing environment, Redis will be used.

Run the `pytest` command in your terminal.

```
(env)$ python -m pytest
```

The tests should fail:

```
django.core.exceptions.ImproperlyConfigured: Requested setting REST_FRAMEWORK, but settings are not
configured.

You must either define the environment variable DJANGO_SETTINGS_MODULE
or call settings.configure() before accessing settings.
```

We need to explicitly identify the Django settings file we want to use.

Let's fix that by creating a `pytest` configuration file. Create a new *pytest.ini* file in the "server" folder with the following code:

```
[pytest]
DJANGO_SETTINGS_MODULE = taxi.settings
```

Run the `pytest` command in your terminal again and you should see another error:

```
ValueError: No application configured for scope type 'websocket'
```

Update the *server/taxi/asgi.py* file like the example below:

```
# server/taxi/asgi.py

import os

from django.core.asgi import get_asgi_application
from django.urls import path # new

from channels.routing import ProtocolTypeRouter, URLRouter # changed

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'taxi.settings')

from trips.consumers import TaxiConsumer

application = ProtocolTypeRouter({
    'http': get_asgi_application(),
    # new
    'websocket': URLRouter([
        path('taxi/', TaxiConsumer.as_asgi()),
    ]),
})
```

Whereas Channels implicitly handles the HTTP URL configuration, we need to explicitly handle WebSocket routing. (A *router* is the Channels counterpart to Django's URL configuration.)

Give the `pytest` command another try. One more error:

```
ModuleNotFoundError: No module named 'trips.consumers'
```

Create a new *server/trips/consumers.py* file with the following code:

```
# server/trips/consumers.py

from channels.generic.websocket import AsyncJsonWebsocketConsumer


class TaxiConsumer(AsyncJsonWebsocketConsumer):
    async def connect(self):
        await self.accept()

    async def disconnect(self, code):
        await super().disconnect(code)
```

A Channels *consumer* is like a Django *view* with extra steps to support the WebSocket protocol. Whereas a Django *view* can only process an incoming request, a Channels *consumer* can send and receive messages and react to the WebSocket connection being opened and closed.

For now, we're explicitly accepting all connections.

Let's run `pytest` one last time to see the tests pass:

```
5 passed in 2.36s
```

## Sending and Receiving Messages

Now that we've gotten our feet wet with opening and closing a connection, let's walk in a little deeper and learn how to send and receive messages.

Add the following test:

```python
# server/trips/tests/test_websocket.py

async def test_can_send_and_receive_messages(self, settings):
    settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
    communicator = WebsocketCommunicator(
        application=application,
        path='/taxi/'
    )
    await communicator.connect()
    message = {
        'type': 'echo.message',
        'data': 'This is a test message.',
    }
    await communicator.send_json_to(message)
    response = await communicator.receive_json_from()
    assert response == message
    await communicator.disconnect()
```

In this test, after we establish a connection with the server, we send a message and wait to get one back. We expect the server to echo our message right back to us exactly the way we sent it. In fact, we need to program this behavior on the server.

Let's open our *server/trips/consumers.py* file and add the following function to the class.

```python
# server/trips/consumers.py

async def receive_json(self, content, **kwargs):
    message_type = content.get('type')
    if message_type == 'echo.message':
        await self.send_json({
            'type': message_type,
            'data': content.get('data'),
        })
```

The `receive_json()` function is responsible for processing all messages that come to the server. Our message is an object with a `type` and a `data` payload. Passing a `type` is a Channels convention that serves two purposes:

1. It helps differentiate incoming messages and tells the server how to process them.
2. The `type` maps directly to a consumer function when sent from another channel layer. (We'll talk about this use in the next section.)

Run `pytest` in your terminal. All tests should pass.

```
6 passed in 3.19s
```

## Sending and Receiving Broadcast Messages

The water is up to our waists now. After this section, we'll be floating.

We saw how to make the client and the server send each other messages through a single instance of an application. Now, let's learn how to make one application talk to another through broadcast messaging.

Add the following test function to our class in *server/tests/test_websocket.py*:

```
# server/trips/tests/test_websocket.py

async def test_can_send_and_receive_broadcast_messages(self, settings):
    settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
    communicator = WebsocketCommunicator(
        application=application,
        path='/taxi/'
    )
    await communicator.connect()
    message = {
        'type': 'echo.message',
        'data': 'This is a test message.',
    }
    channel_layer = get_channel_layer()
    await channel_layer.group_send('test', message=message)
    response = await communicator.receive_json_from()
    assert response == message
    await communicator.disconnect()
```

Add to the top:

```
from channels.layers import get_channel_layer
```

This new test looks a lot like the last test we wrote, but it has one important difference: It uses a *channel layer* to broadcast a message to a *group*. Whereas the last test modeled a user talking to himself in an empty room, this most recent test represents a user talking to a room full of people.

We need to modify our consumer in two ways to get our newest test passing. Add the code as shown below:

```
# server/trips/consumers.py

from channels.generic.websocket import AsyncJsonWebsocketConsumer


class TaxiConsumer(AsyncJsonWebsocketConsumer):
    groups = ['test'] # new

    async def connect(self):
        await self.accept()

    async def disconnect(self, code):
        await super().disconnect(code)

    async def echo_message(self, message): # new
        await self.send_json({
            'type': message.get('type'),
            'data': message.get('data'),
        })

    async def receive_json(self, content, **kwargs):
        message_type = content.get('type')
        if message_type == 'echo.message':
            await self.send_json({
                'type': message_type,
                'data': content.get('data'),
            })
```

As stated, the message `type` maps to a consumer function. That statement is true when we're talking about messages that are broadcast to groups. When a message comes from a channel layer, Channels looks for a function on the receiving consumer whose name matches the message `type`. It also converts any `.` characters to `_` characters before it checks for the match.

Channel layers broadcast messages to specific groups, which are collections of other channel layers that are subscribed to the same topic. One way to subscribe to a group is by defining the membership in a class variable like the change we added above.

Make the following change to keep the same behavior while making group subscription more explicit:

```python
# server/trips/consumers.py

from channels.generic.websocket import AsyncJsonWebsocketConsumer


class TaxiConsumer(AsyncJsonWebsocketConsumer):
    groups = ['test']

    async def connect(self): # changed
        await self.channel_layer.group_add(
            group='test',
            channel=self.channel_name
        )
        await self.accept()

    async def disconnect(self, code): # changed
        await self.channel_layer.group_discard(
            group='test',
            channel=self.channel_name
        )
        await super().disconnect(code)

    async def echo_message(self, message):
        await self.send_json({
            'type': message.get('type'),
            'data': message.get('data'),
        })

    async def receive_json(self, content, **kwargs):
        message_type = content.get('type')
        if message_type == 'echo.message':
            await self.send_json({
                'type': message_type,
                'data': content.get('data'),
            })
```

With these changes, any client connected to the `TaxiConsumer` through WebSockets will automatically be subscribed to the `test` group. When a channel layer sends a broadcast message with the type `echo.message`, Channels will execute the `echo_message()` function for everyone in the `test` group.

Run `pytest` and confirm that the tests pass.

```
7 passed in 2.37s
```

## Authenticating Socket Connections

Establishing a WebSocket connection starts with a "handshake" between the client and the server over HTTP. Anything that can be sent with an HTTP request can be sent with the handshake -- i.e., headers and cookies, query string parameters, and request bodies. Unfortunately, the JavaScript WebSocket API does not support custom headers. That means we need to find a different way to authenticate our WebSocket connection than an authorization header.

We have several different ways to get around the custom headers limitation, but the community at large seems to agree that sending the access token in a query string parameter is the way to go. Keep in mind that in a production environment, you need to be careful to protect the access token from bad actors.

Let's write a test to show that a connection fails if the handshake request does not include a valid access token.

```
# server/trips/tests/test_websocket.py

async def test_cannot_connect_to_socket(self, settings):
    settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
    communicator = WebsocketCommunicator(
        application=application,
        path='/taxi/'
    )
    connected, _ = await communicator.connect()
    assert connected is False
    await communicator.disconnect()
```

Create a new *server/taxi/middleware.py* file with the following code:

```
# server/taxi/middleware.py

from urllib.parse import parse_qs

from django.contrib.auth import get_user_model
from django.contrib.auth.models import AnonymousUser
from django.db import close_old_connections
from channels.auth import AuthMiddleware
from channels.db import database_sync_to_async
from channels.sessions import CookieMiddleware, SessionMiddleware
from rest_framework_simplejwt.tokens import AccessToken

User = get_user_model()


@database_sync_to_async
def get_user(scope):
    close_old_connections()
    query_string = parse_qs(scope['query_string'].decode())
    token = query_string.get('token')
    if not token:
        return AnonymousUser()
    try:
        access_token = AccessToken(token[0])
        user = User.objects.get(id=access_token['id'])
    except Exception as exception:
        return AnonymousUser()
    if not user.is_active:
        return AnonymousUser()
    return user


class TokenAuthMiddleware(AuthMiddleware):
    async def resolve_scope(self, scope):
        scope['user']._wrapped = await get_user(scope)


def TokenAuthMiddlewareStack(inner):
    return CookieMiddleware(SessionMiddleware(TokenAuthMiddleware(inner)))
```

Our new middleware class plucks the JWT access token from the query string and retrieves the associated user. Once the WebSocket connection is opened, all messages can be sent and received without verifying the user again. Closing the connection and opening it again requires re-authorization.

Let's implement the middleware. Open the *server/taxi/asgi.py* file and make the following changes:

```
# server/taxi/asgi.py

import os

from django.core.asgi import get_asgi_application
from django.urls import path

from channels.routing import ProtocolTypeRouter, URLRouter

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'taxi.settings')

from taxi.middleware import TokenAuthMiddlewareStack # new
from trips.consumers import TaxiConsumer

application = ProtocolTypeRouter({
    'http': get_asgi_application(),
    'websocket': TokenAuthMiddlewareStack( # changed
        URLRouter([
            path('taxi/', TaxiConsumer.as_asgi()),
        ])
    ),
})
```

Here, we're wrapping our URL router in our middleware stack, so all incoming connection requests will go through our authentication method.

With the middleware in place, let's edit our consumer to reject any connection that does not have an authenticated user.

Edit the `connect()` function like so:

```
# server/trips/consumers.py

async def connect(self): # changed
    user = self.scope['user']
    if user.is_anonymous:
        await self.close()
    else:
        await self.channel_layer.group_add(
            group='test',
            channel=self.channel_name
        )
        await self.accept()
```

We need to add one more `mark` to our `pytest` test class in order to access the database.

```
# server/trips/tests/test_websocket.py

@pytest.mark.asyncio
@pytest.mark.django_db(transaction=True) # new
class TestWebSocket: ...
```

Run the `pytest` command in your terminal.

```
1 failed, 7 passed in 3.22s
```

We need to refactor our other WebSocket tests to pass a JWT access token in the query string when connecting. Add the following `create_user()` helper function after `TEST_CHANNEL_LAYERS` and before `TestWebSocket`:

```
# server/trips/tests/test_websocket.py

@database_sync_to_async
def create_user(username, password):
    user = get_user_model().objects.create_user(
        username=username,
        password=password
    )
    access = AccessToken.for_user(user)
    return user, access
```

We also need to add the following imports to the top of the page:

```
from channels.db import database_sync_to_async
from django.contrib.auth import get_user_model
from rest_framework_simplejwt.tokens import AccessToken
```

Our helper function creates a new user in the database and then generates an access token for it.

Edit the first test to match the code below:

```
# server/trips/tests/test_websocket.py

async def test_can_connect_to_server(self, settings):
    settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
    _, access = await create_user(  # new
        'test.user@example.com', 'pAssw0rd'
    )
    communicator = WebsocketCommunicator(
        application=application,
        path=f'/taxi/?token={access}' # changed
    )
    connected, _ = await communicator.connect()
    assert connected is True
    await communicator.disconnect()
```

Make the same change to the next two functions -- e.g., call the `create_user()` function to get the access token and then pass it as a query string parameter in the communicator's `path`.

Final code:

```python
# server/trips/tests/test_websocket.py

import pytest
from channels.db import database_sync_to_async
from channels.layers import get_channel_layer
from channels.testing import WebsocketCommunicator
from django.contrib.auth import get_user_model
from rest_framework_simplejwt.tokens import AccessToken

from taxi.asgi import application

TEST_CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels.layers.InMemoryChannelLayer',
    },
}


@database_sync_to_async
def create_user(username, password):
    user = get_user_model().objects.create_user(
        username=username,
        password=password
    )
    access = AccessToken.for_user(user)
    return user, access


@pytest.mark.asyncio
@pytest.mark.django_db(transaction=True)
class TestWebSocket:
    async def test_can_connect_to_server(self, settings):
        settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
        _, access = await create_user(
            'test.user@example.com', 'pAssw0rd'
        )
        communicator = WebsocketCommunicator(
            application=application,
            path=f'/taxi/?token={access}'
        )
        connected, _ = await communicator.connect()
        assert connected is True
        await communicator.disconnect()

    async def test_can_send_and_receive_messages(self, settings):
        settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
        _, access = await create_user(
            'test.user@example.com', 'pAssw0rd'
        )
        communicator = WebsocketCommunicator(
            application=application,
            path=f'/taxi/?token={access}'
        )
        await communicator.connect()
        message = {
            'type': 'echo.message',
            'data': 'This is a test message.',
        }
        await communicator.send_json_to(message)
        response = await communicator.receive_json_from()
        assert response == message
        await communicator.disconnect()

    async def test_can_send_and_receive_broadcast_messages(self, settings):
        settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
        _, access = await create_user(
            'test.user@example.com', 'pAssw0rd'
        )
        communicator = WebsocketCommunicator(
            application=application,
            path=f'/taxi/?token={access}'
        )
        await communicator.connect()
        message = {
            'type': 'echo.message',
            'data': 'This is a test message.',
        }
        channel_layer = get_channel_layer()
        await channel_layer.group_send('test', message=message)
        response = await communicator.receive_json_from()
        assert response == message
        await communicator.disconnect()

    async def test_cannot_connect_to_socket(self, settings):
        settings.CHANNEL_LAYERS = TEST_CHANNEL_LAYERS
        communicator = WebsocketCommunicator(
            application=application,
            path='/taxi/'
        )
        connected, _ = await communicator.connect()
        assert connected is False
        await communicator.disconnect()
```

Run `pytest` to see the tests passing.

```
8 passed in 3.83s
```

Now that you know the basics, we're ready to build on that knowledge in the next chapter.

Confirm that your directory looks like this before moving on:

```
└── server
    ├── manage.py
    ├── pytest.ini
    ├── taxi
    │   ├── __init__.py
    │   ├── asgi.py
    │   ├── middleware.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    └── trips
        ├── __init__.py
        ├── admin.py
        ├── apps.py
        ├── consumers.py
        ├── migrations
        │   ├── 0001_initial.py
        │   ├── 0002_trip.py
        │   └── __init__.py
        ├── models.py
        ├── serializers.py
        ├── tests
        │   ├── __init__.py
        │   ├── test_http.py
        │   └── test_websocket.py
        ├── urls.py
        └── views.py
```

« HTTP

✓ Mark as Completed