¡Perfecto! Vamos desde cero. Es la mejor manera de aprender.

Viendo la estructura de tu proyecto (app/api, app/services, app/schemas), ya tienes una arquitectura excelente y limpia. Vamos a usarla.

La guía será paso a paso. No pases al siguiente paso hasta que el anterior esté completo y lo entiendas.

---

## Prerrequisito: Ajustar tu Base de Datos

Antes de escribir una sola línea de Python, tenemos que resolver un problema. Tu endpoint debe devolver un estimatedArrivalTime (hora de llegada estimada).

Para que el servidor pueda calcular esto y *recordar* el progreso del viaje, tu tabla ships **necesita más columnas**.

La tabla que me mostraste:
id, owner_id, is_moving, current_pos_x, current_pos_y, start_pos_x, start_pos_y, end_pos_x, end_pos_y
**Necesitas añadirle:**

- movement_start_time (de tipo timestamp o datetime)
- estimated_arrival_time (de tipo timestamp o datetime)
- speed (de tipo float8. ¿O todas las naves tienen la misma velocidad?)

**Pregunta 1: ¿Sabes cómo modificar tu modelo de SQLAlchemy (probablemente en app/models/ship.py) para añadir estas columnas?**

No sigas hasta que tengas esto. Sería algo así en app/models/ship.py:

Python

```
# En app/models/ship.py (o donde definas tu tabla 'ships')
from sqlalchemy import Column, String, Boolean, Float, DateTime, ForeignKey
from sqlalchemy.dialects.postgresql import UUID
from ..db.base_class import Base # Asumo que tienes una base
import datetime

class Ship(Base):
```

```
    __tablename__ = "ships"

    id = Column(UUID(as_uuid=True), primary_key=True, index=True)
    owner_id = Column(UUID(as_uuid=True), ForeignKey("users.id"), unique=True, index=True) # Asumo que se liga a 'users'

    is_moving = Column(Boolean, default=False)

    # Posición actual (la que actualiza el 'game tick')
    current_pos_x = Column(Float, default=0.0)
    current_pos_y = Column(Float, default=0.0)

    # Datos del viaje
    start_pos_x = Column(Float, nullable=True)
    start_pos_y = Column(Float, nullable=True)
    end_pos_x = Column(Float, nullable=True)
    end_pos_y = Column(Float, nullable=True)

    # ¡¡NUEVAS COLUMNAS!!
    movement_start_time = Column(DateTime, nullable=True)
    estimated_arrival_time = Column(DateTime, nullable=True)

    speed = Column(Float, default=100.0) # Velocidad base de la nave (unidades/segundo)
```

*(Una vez que modifiques esto, puede que necesites regenerar tus tablas o usar una migración como Alembic).*

---

## Paso 1: Definir los "Contratos" (Schemas)

Los "schemas" (con Pydantic) definen qué datos entran y qué datos salen de tu API. Ya tienes app/schemas/user.py, así que vamos a crear app/schemas/ship.py.

**Tu Tarea:** Crea el archivo app/schemas/ship.py y añade el siguiente contenido.

Python

```
# En app/schemas/ship.py
```

```python
from pydantic import BaseModel
from datetime import datetime

# --- Schemas para el movimiento ---

# Un modelo base para coordenadas
class Position(BaseModel):
    x: float
    y: float

# 1. Lo que el jugador ENVÍA (Request Body)
class ShipMoveRequest(BaseModel):
    targetPosition: Position

# 2. Los datos que la API DEVUELVE (anidado en la respuesta)
class ShipMoveResponseData(BaseModel):
    endPosition: Position
    estimatedArrivalTime: datetime

# 3. La respuesta COMPLETA de la API
class ShipMoveResponse(BaseModel):
    status: str = "success"
    message: str = "Movement initiated."
    data: ShipMoveResponseData
```

**Pregunta 2: ¿Entiendes por qué separamos Position, ShipMoveRequest y ShipMoveResponse?**

- ShipMoveRequest define la *entrada*.
- ShipMoveResponse define la *salida*.
- Position es un modelo reutilizable.

---

## Paso 2: Crear el Endpoint (La "Ruta" API)

Aquí es donde creamos el endpoint POST /api/v1/player/move. Basado en tu estructura, esto debería ir en un archivo nuevo.

**Tu Tarea:** Crea el archivo app/api/ship.py y añade esto.

Python

```python
# En app/api/ship.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
import datetime

from ... import schemas # Importa tus schemas del Paso 1
from ... import services # Importará el servicio del Paso 3
from ... import models # Importará tus modelos de BD
from ..deps import get_db, get_current_user # ¡¡MUY IMPORTANTE!!

router = APIRouter()

@router.post(
    "/move",
    response_model=schemas.ShipMoveResponse,
    summary="Iniciar Movimiento de la Nave"
)
async def move_ship(
    *,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user), # Asumo que tienes esto
    move_request: schemas.ShipMoveRequest # El body de la petición
):
    """
    Establece un nuevo punto de destino para la nave del jugador.
    """

    # ---------------------------------------------------------------
    # ¡AQUÍ IRÁ LA LÓGICA EN EL PASO 4!
    # Por ahora, solo devolvemos datos falsos para probar
    # ---------------------------------------------------------------

    # Datos falsos temporales
    fake_eta = datetime.datetime.utcnow() + datetime.timedelta(seconds=60)

    return {
        "status": "success",
        "message": "Movement initiated.",
        "data": {
            "endPosition": move_request.targetPosition,
            "estimatedArrivalTime": fake_eta
        }
```

```
}
```

**Pregunta 3 (¡La más importante!):** ¿Tienes un sistema de dependencias de autenticación?

En el código de arriba, usé Depends(get_current_user). Esto asume que tienes un archivo (quizás app/api/deps.py) que se encarga de:

1. Leer el token JWT de la cabecera.
2. Validarlo.
3. Extraer el ID de usuario y devolver el objeto User de la base de datos.

Si no tienes esto, **no puedes** identificar *qué* nave mover.

---

## Paso 3: La Lógica de Negocio (El "Servicio")

El endpoint (api/ship.py) no debe hacer cálculos. Solo recibe la petición y la pasa a un "servicio". Aquí es donde ocurre la magia.

**Tu Tarea:** Crea (o edita) el archivo app/services/ship.py.

Python

```python
# En app/services/ship.py
from sqlalchemy.orm import Session
from .. import models, schemas
import math
import datetime

# Constante de velocidad (o la sacas del 'ship.speed' de la BD)
SHIP_SPEED_UNITS_PER_SEC = 100.0

def start_player_move(
    db: Session,
    user_id: str, # Viene del token JWT
    target_pos: schemas.Position
) -> schemas.ShipMoveResponseData:
```

```
    # 1. Encontrar la nave del jugador
    ship = db.query(models.Ship).filter(models.Ship.owner_id == user_id).first()

    if not ship:
        # Esto es un error, el endpoint lo capturará
        raise Exception("Ship not found")
        # (Mejor si creas una excepción personalizada)

    # 2. Definir variables de inicio
    start_time = datetime.datetime.utcnow()
    start_pos = schemas.Position(x=ship.current_pos_x, y=ship.current_pos_y)

    # 3. Calcular distancia y duración
    # Distancia = sqrt((x2-x1)^2 + (y2-y1)^2)
    distance = math.sqrt(
        (target_pos.x - start_pos.x) ** 2 +
        (target_pos.y - start_pos.y) ** 2
    )

    if distance == 0:
        # Ya está en el destino, o es un clic inválido
        raise Exception("Already at target position or invalid distance")

    duration_seconds = distance / SHIP_SPEED_UNITS_PER_SEC # Usa ship.speed si es variable

    # 4. Calcular hora de llegada
    eta = start_time + datetime.timedelta(seconds=duration_seconds)

    # 5. ACTUALIZAR la nave en la BD
    ship.is_moving = True
    ship.start_pos_x = start_pos.x
    ship.start_pos_y = start_pos.y
    ship.end_pos_x = target_pos.x
    ship.end_pos_y = target_pos.y
    ship.movement_start_time = start_time
    ship.estimated_arrival_time = eta

    db.commit()
    db.refresh(ship)

    # 6. Devolver los datos para la respuesta
    return schemas.ShipMoveResponseData(
        endPosition=target_pos,
```

```
        estimatedArrivalTime=eta
    )
```

---

## Paso 4: Conectar el Endpoint (Paso 2) con el Servicio (Paso 3)

Ahora, volvemos a app/api/ship.py y reemplazamos los datos falsos por la llamada real al servicio.

**Tu Tarea:** Edita app/api/ship.py que creaste en el Paso 2.

Python

```python
# En app/api/ship.py (¡ACTUALIZADO!)
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
import datetime

from ... import schemas
from ... import services # ¡¡Asegúrate de importar services!!
from ... import models
from ..deps import get_db, get_current_user

router = APIRouter()

@router.post(
    "/move",
    response_model=schemas.ShipMoveResponse,
    summary="Iniciar Movimiento de la Nave"
)
async def move_ship(
    *,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_user),
    move_request: schemas.ShipMoveRequest
):
    """
    Establece un nuevo punto de destino para la nave del jugador.
```

```
    """

    try:
        # ¡¡AQUÍ ESTÁ LA CONEXIÓN!!
        response_data = services.ship.start_player_move(
            db=db,
            user_id=current_user.id, # Asumo que 'current_user' tiene 'id'
            target_pos=move_request.targetPosition
        )

        return {
            "status": "success",
            "message": "Movement initiated.",
            "data": response_data # Devuelve los datos del servicio
        }

    except Exception as e:
        # Manejo de errores simple
        raise HTTPException(
            status_code=400,
            detail=f"Error starting movement: {str(e)}"
        )
```

---

## Paso 5: Incluir el Router en tu App Principal

Finalmente, tienes que decirle a tu main.py que existe este nuevo grupo de endpoints.

**Tu Tarea:** Edita tu app/main.py.

Python

```
# En app/main.py (extracto)
from fastapi import FastAPI
from .api import user, auth # Asumo que tienes estos
from .api import ship # ¡¡EL NUEVO!!
```

```python
app = FastAPI()

# ... otros routers ...
app.include_router(auth.router, prefix="/api/v1/auth", tags=["Auth"])
app.include_router(user.router, prefix="/api/v1/users", tags=["Users"])

# ¡¡AÑADIR ESTA LÍNEA!!
app.include_router(
    ship.router,
    prefix="/api/v1/player", # El prefijo que pediste
    tags=["Player"]
)

@app.get("/")
def read_root():
    return {"Hello": "World"}
```

---

## Resumen

Si seguiste los 5 pasos, ahora tienes:

1. **Modelos (models/ship.py)** actualizados con las columnas de tiempo.
2. **Schemas (schemas/ship.py)** que definen la entrada y salida del endpoint.
3. **Servicio (services/ship.py)** que hace todos los cálculos y actualiza la BD.
4. **Endpoint (api/ship.py)** que maneja la autenticación, recibe el JSON, llama al servicio y devuelve la respuesta.
5. **App Principal (main.py)** que publica tu nuevo endpoint en la URL POST /api/v1/player/move.

**Vuelve a la Pregunta 3:** El punto más crítico es Depends(get_current_user). ¿Ya tienes esa dependencia de autenticación funcionando?