### Using Replit to Develop, Test, and Deploy Your FastAPI Application

Replit is an excellent platform for developing, testing, and deploying FastAPI applications due to its simplicity and online accessibility. Follow these steps to set up and run your FastAPI application on Replit:

### 1. Setting Up Your Project on Replit

1. **Create a New Replit**:
   - Go to [Replit](https://replit.com/) and create a new repl.
   - Choose "Python" as the language.

2. **Install FastAPI and Uvicorn**:
   - In the Replit shell, install FastAPI and Uvicorn by running:
     ```bash
     pip install fastapi uvicorn
     ```

3. **Add Your FastAPI Code**:
   - Replace the contents of `main.py` with your integrated FastAPI application code. Ensure the code is adapted to work within Replit's environment.

### 2. Your FastAPI Application Code for Replit

Here's the FastAPI application code, optimized for Replit:

```python
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel
from typing import List, Optional

app = FastAPI(
    title="Space Age Hustle: Oneseco's Orbit API",
    description="API for managing the interactive game 'Space Age Hustle: Oneseco's Orbit'",
    version="1.0.1"
)

# Pydantic models for game settings and state
class GameSettings(BaseModel):
    story_name: str
    background_image_url: str
    song_playlist_urls: List[str]

class Song(BaseModel):
```

```
    url: str
    title: str
    artist: str

class GameState(BaseModel):
    location: str
    health: int
    inventory: List[str]
    objectives: List[str]

class CharacterInteraction(BaseModel):
    character_name: str
    dialogue_option: str

class CharacterInteractionResponse(BaseModel):
    response_text: str

class ExploreRequest(BaseModel):
    area: str

class ExploreResponse(BaseModel):
    description: str
    items_found: List[str]

# Pydantic models for character creator
class Character(BaseModel):
    name: str
    role: str
    background: str
    physical_description: str
    distinctive_features: str
    primary_skill: str
    secondary_skills: List[str]
    weaknesses: List[str]
    primary_weapon: str
    additional_gear: str
    personality_traits: str
    motivations: str
    allies_and_enemies: str
    primary_goal: str
    ambitions: str

# Pydantic models for interactive fiction
class Choice(BaseModel):
```

```
    text: str
    next_scene_id: Optional[int]

class Scene(BaseModel):
    id: int
    title: str
    description: str
    choices: List[Choice]

# In-memory storage for game state, settings, characters, and scenes
game_settings = None
game_state = GameState(location="Atmos Docks", health=100, inventory=["Repair Kit", "Energy
Cell"], objectives=["Fix the craft", "Complete paperwork"])
playlist = []
characters = []
scenes = []

# Endpoints for game management
@app.post("/game/start", response_model=GameSettings)
async def start_game(settings: GameSettings):
    global game_settings
    game_settings = settings
    return game_settings

@app.get("/game/settings", response_model=GameSettings)
async def fetch_game_settings():
    if game_settings is None:
        raise HTTPException(status_code=404, detail="Game settings not found")
    return game_settings

@app.post("/playlist", response_model=Song)
async def add_song_to_playlist(song: Song):
    playlist.append(song)
    return song

@app.delete("/playlist", response_model=dict)
async def remove_song_from_playlist(url: str = Query(...)):
    global playlist
    playlist = [song for song in playlist if song.url != url]
    return {"message": "Song removed successfully"}

@app.get("/game/state", response_model=GameState)
async def get_game_state():
    return game_state
```

```
@app.post("/game/state", response_model=GameState)
async def update_game_state(state: GameState):
    global game_state
    game_state = state
    return game_state

@app.post("/character/interact", response_model=CharacterInteractionResponse)
async def interact_with_character(interaction: CharacterInteraction):
    response_text = f"You interacted with {interaction.character_name} and chose the option:
{interaction.dialogue_option}"
    return CharacterInteractionResponse(response_text=response_text)

@app.post("/explore", response_model=ExploreResponse)
async def explore_area(request: ExploreRequest):
    description = f"You explore the area: {request.area}"
    items_found = ["Ancient Tech", "Mystery Artifact"]
    return ExploreResponse(description=description, items_found=items_found)

# Endpoints for character creator
@app.post("/characters", response_model=Character)
async def create_character(character: Character):
    characters.append(character)
    return character

@app.get("/characters", response_model=List[Character])
async def get_characters():
    return characters

@app.get("/characters/{name}", response_model=Character)
async def get_character(name: str):
    for character in characters:
        if character.name == name:
            return character
    raise HTTPException(status_code=404, detail="Character not found")

@app.put("/characters/{name}", response_model=Character)
async def update_character(name: str, updated_character: Character):
    for index, character in enumerate(characters):
        if character.name == name:
            characters[index] = updated_character
            return updated_character
    raise HTTPException(status_code=404, detail="Character not found")
```

```
@app.delete("/characters/{name}", response_model=dict)
async def delete_character(name: str):
    global characters
    characters = [character for character in characters if character.name != name]
    return {"message": "Character deleted successfully"}

# Endpoints for interactive fiction
@app.post("/scenes", response_model=Scene)
async def create_scene(scene: Scene):
    scenes.append(scene)
    return scene

@app.get("/scenes", response_model=List[Scene])
async def get_scenes():
    return scenes

@app.get("/scenes/{scene_id}", response_model=Scene)
async def get_scene(scene_id: int):
    for scene in scenes:
        if scene.id == scene_id:
            return scene
    raise HTTPException(status_code=404, detail="Scene not found")

@app.put("/scenes/{scene_id}", response_model=Scene)
async def update_scene(scene_id: int, updated_scene: Scene):
    for index, scene in enumerate(scenes):
        if scene.id == scene_id:
            scenes[index] = updated_scene
            return updated_scene
    raise HTTPException(status_code=404, detail="Scene not found")

@app.delete("/scenes/{scene_id}", response_model=dict)
async def delete_scene(scene_id: int):
    global scenes
    scenes = [scene for scene in scenes if scene.id != scene_id]
    return {"message": "Scene deleted successfully"}

# Run the server using Uvicorn
if __name__ == '__main__':
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

### 3. Configuring the `replit.nix` File

Replit uses a `replit.nix` file to manage dependencies and the environment. You need to create or edit this file to ensure FastAPI and Uvicorn are installed. Here's an example `replit.nix` configuration:

```nix
{ pkgs }: {
    deps = [
        pkgs.python39Full
        pkgs.python39Packages.pip
        pkgs.python39Packages.fastapi
        pkgs.python39Packages.uvicorn
    ];
}
```

### 4. Running the Server

In Replit, add a new file called `.replit` and configure it to run the Uvicorn server automatically:

```ini
run = "uvicorn main:app --host 0.0.0.0 --port 8000 --reload"
```

### 5. Starting Your Application

- Click the "Run" button in Replit. This will start your FastAPI application.
- You can now interact with your API using the Replit web interface or through external tools like Postman.

### 6. Testing and Iterating

- **Manual Testing**: Use Replit's built-in web view or an external API testing tool to manually test each endpoint.
- **Continuous Development**: Make changes to `main.py` and other files as needed. Each change will automatically trigger a reload of your FastAPI server.

### 7. Persistent Storage (Optional)

If you need persistent storage (e.g., a database), consider integrating with Replit's database or an external database service. For initial development, you can continue using in-memory storage.

This setup provides a convenient and accessible environment for developing, testing, and running your FastAPI application.