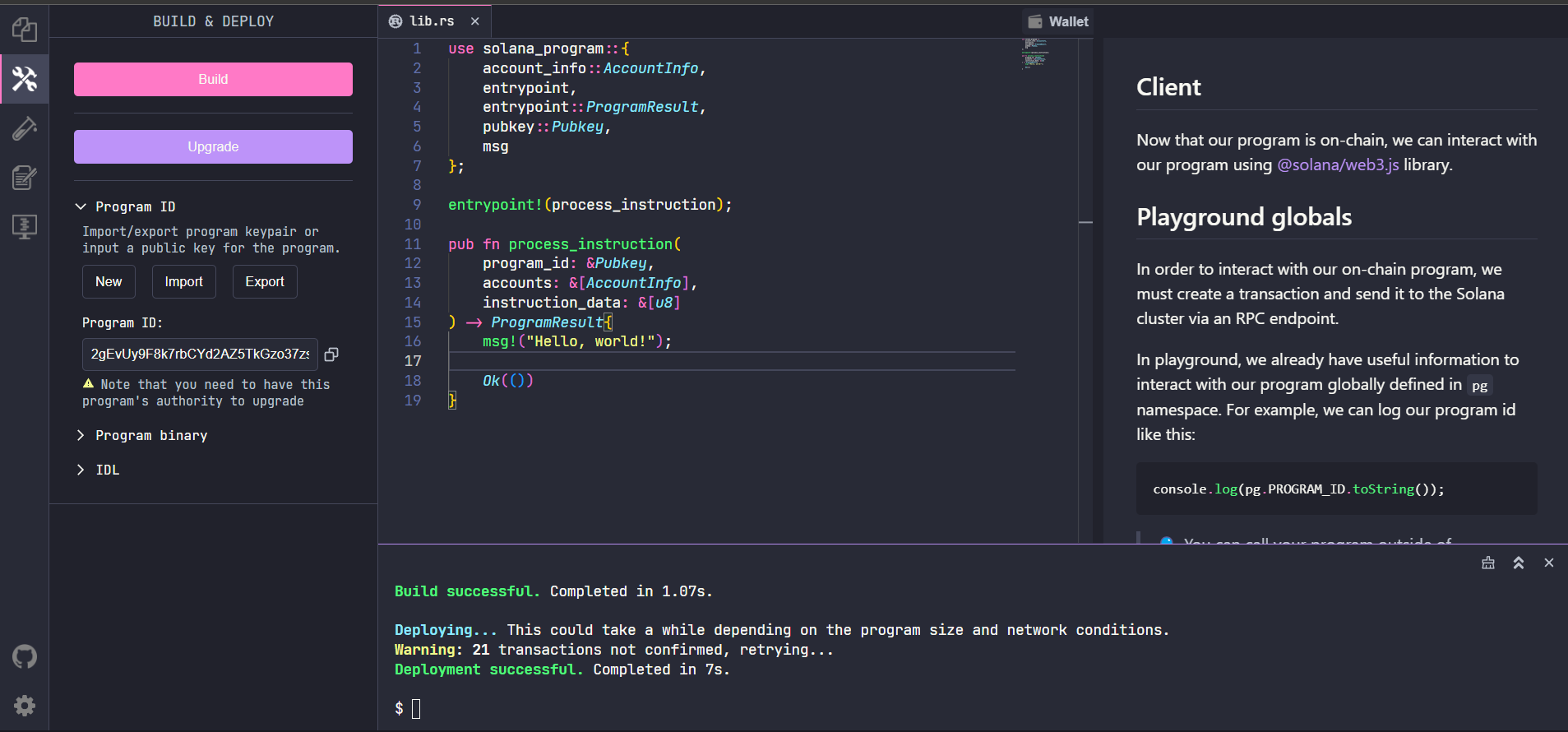
Module 2

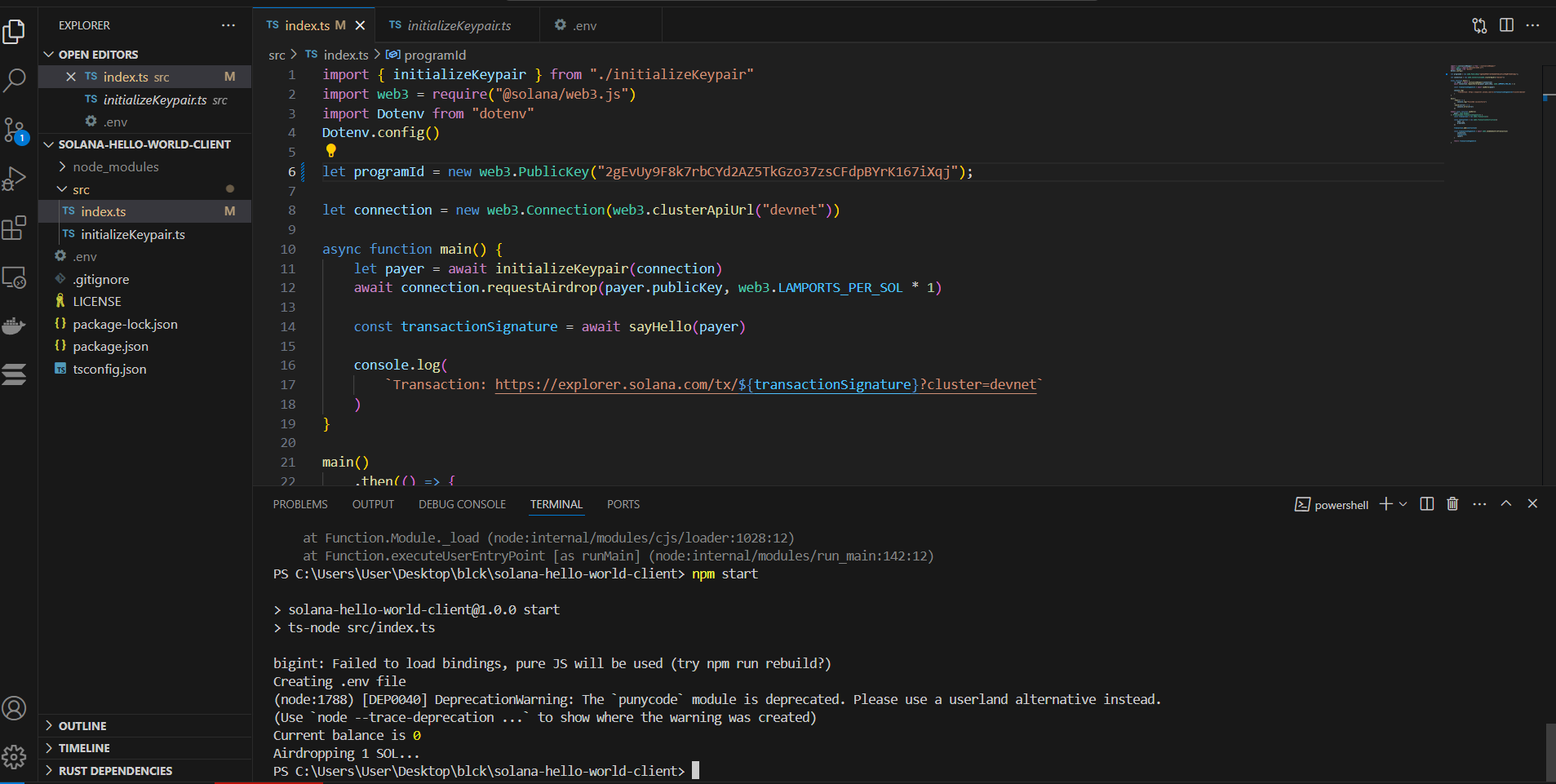
1. Hello World!

Open solana playhround and paste code from a lesson on SolDev.

Connect your wallet

Build and Deploy





**Handle Instruction Data**

we'll focus on deserializing the instruction data. The following lesson will focus on the second half of this program.

1. Entry point

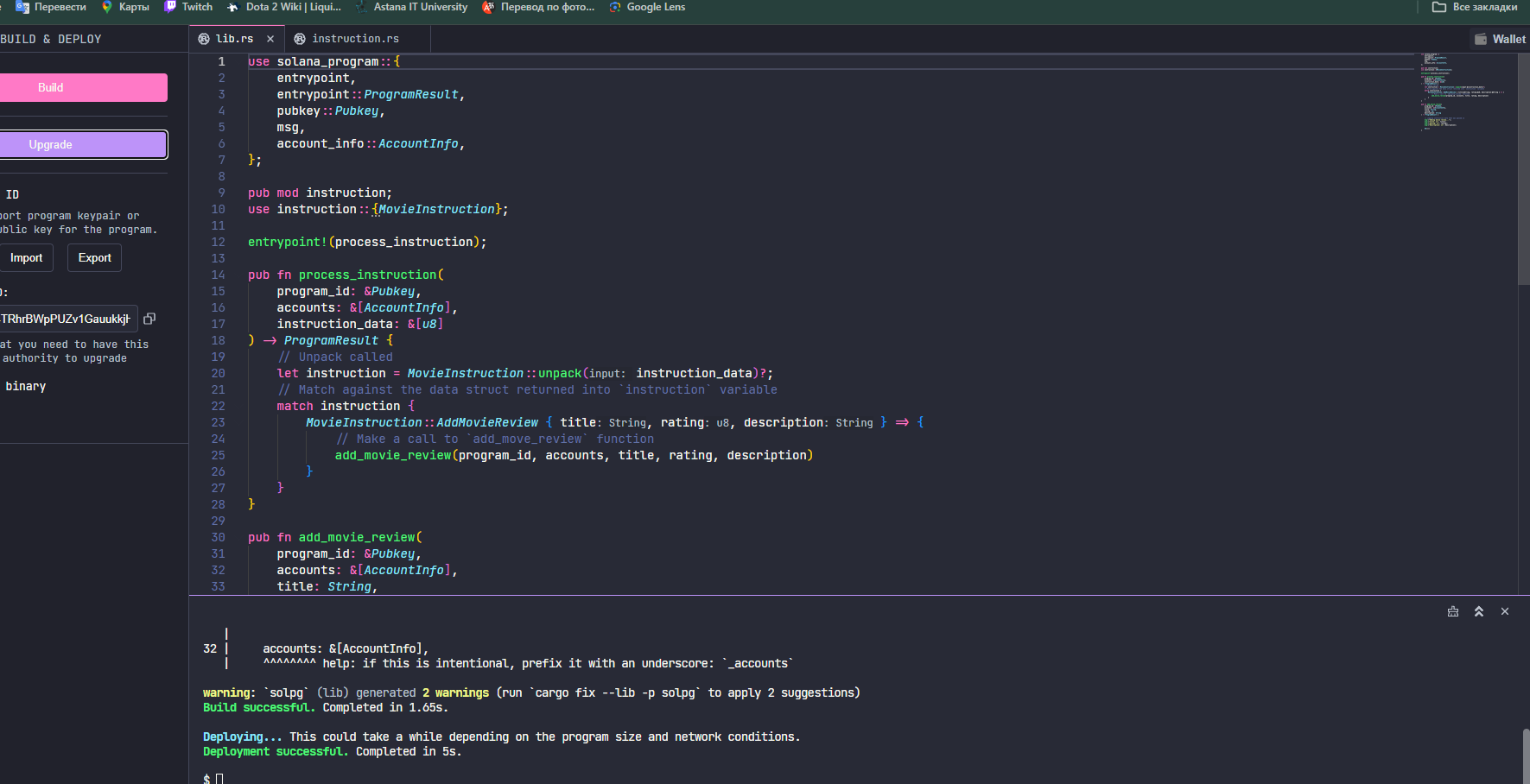
We’ll be using Solana Playground again to build out this program.

2. Deserialize instruction data

Before we continue with the processor logic, we should define our supported instructions and implement our deserialization function.

3. Program logic

With the instruction deserialization handled, we can return to the lib.rs file to handle some of our program logic.



**State** **Management**

1. Get the starter code

Our program currently includes the instruction.rs file we use to deserialize the instruction\_data passed into the program entry point. We have also completed lib.rs file to the point where we can print our deserialized instruction data to the program log using the msg! macro.

2. Create struct to represent account data

Let’s begin by creating a new file named state.rs.

This file will:

Define the struct our program uses to populate the data field of a new account

Add BorshSerialize and BorshDeserialize traits to this struct

First, let’s bring into scope everything we’ll need from the borsh crate.

use borsh::{BorshSerialize, BorshDeserialize};

Next, let’s create our MovieAccountState struct. This struct will define the parameters that each new movie review account will store in its data field. Our MovieAccountState struct will require the following parameters:

is\_initialized - shows whether or not the account has been initialized

rating - user’s rating of the movie

description - user’s description of the movie

title - title of the movie the user is reviewing

Next, let’s continue building out our add\_movie\_review function. Recall that an array of accounts is passed into the add\_movie\_review function through a single accounts argument. To process our instruction, we will need to iterate through accounts and assign the AccountInfo for each account to its own variable.

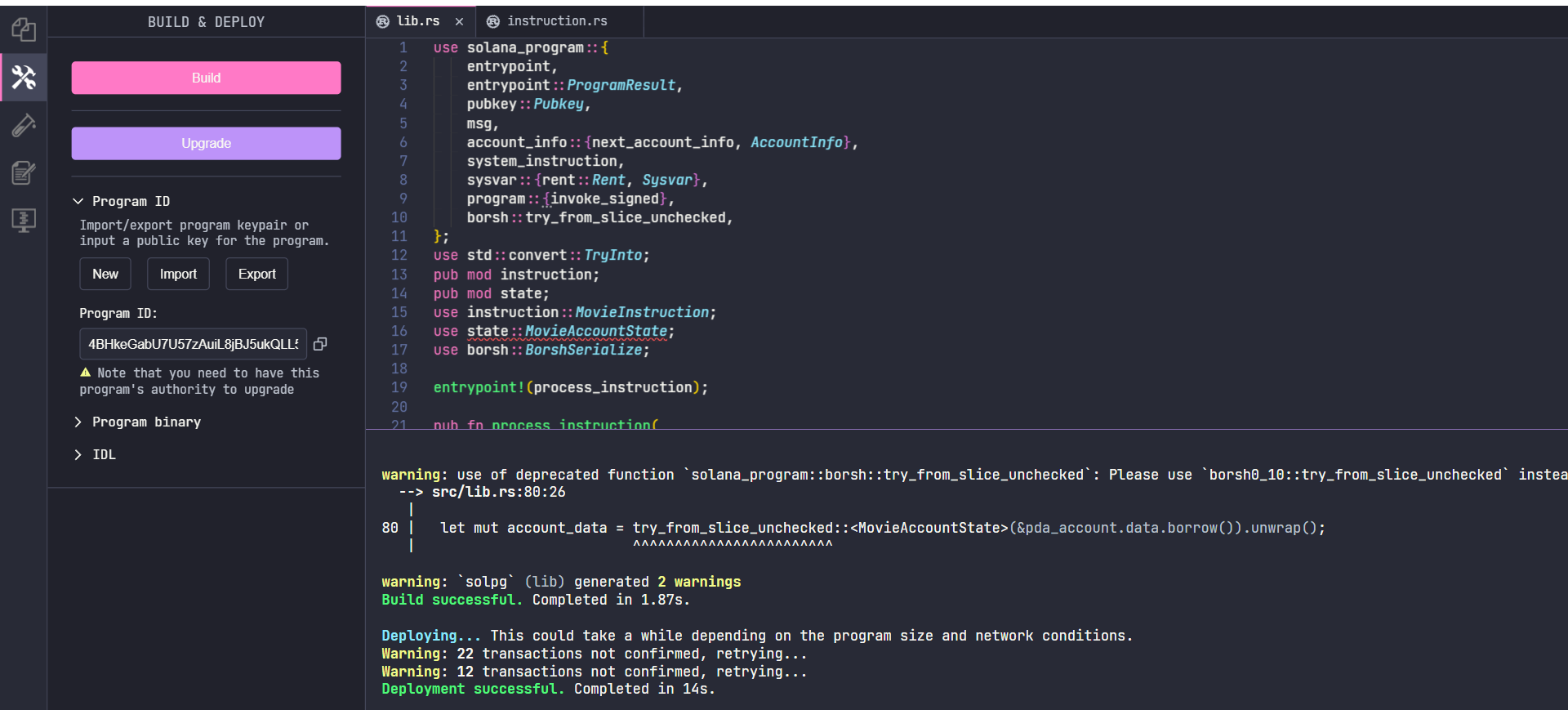
Next, within our add\_movie\_review function, let’s independently derive the PDA we expect the user to have passed in. We'll need to provide the bump seed for the derivation later, so even though pda\_account should reference the same account, we still need to call find\_program\_address.

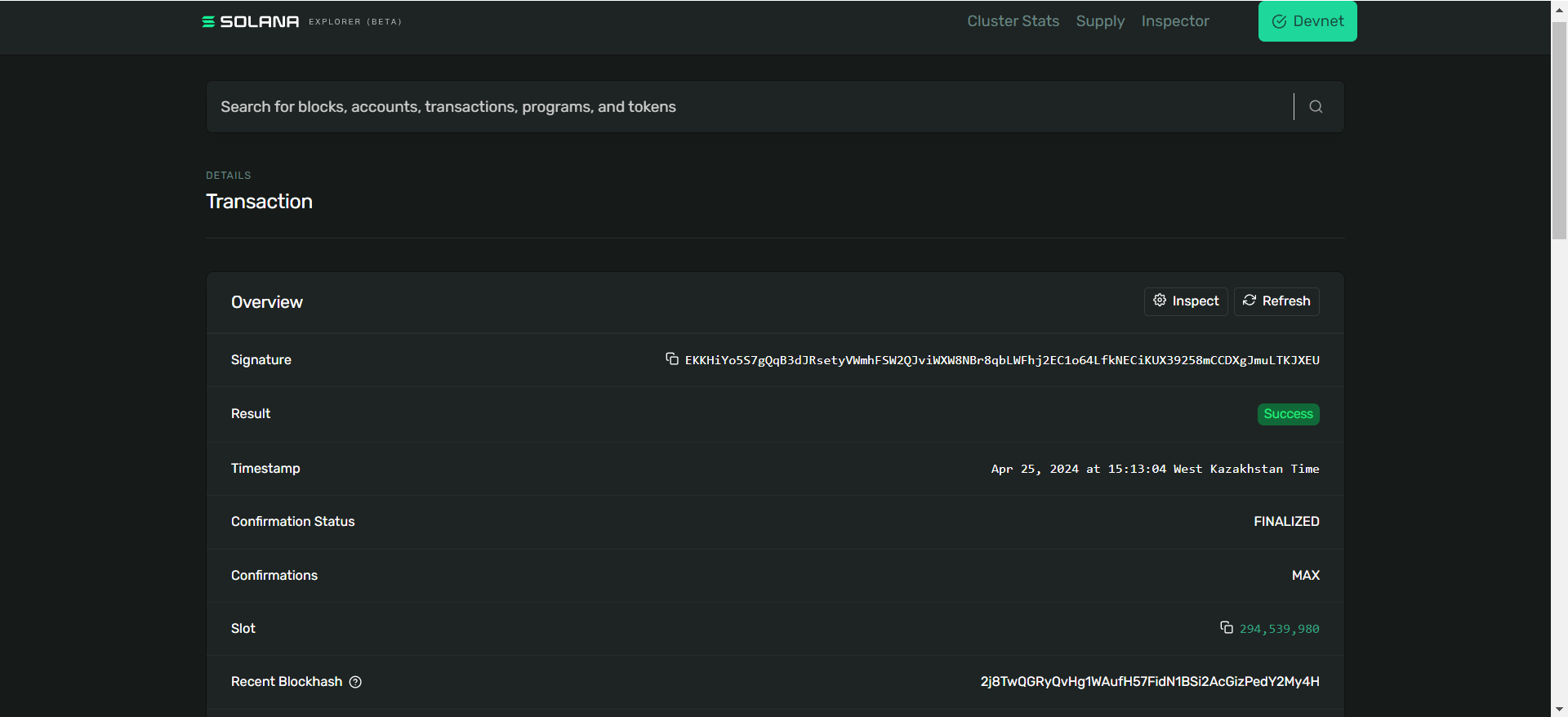
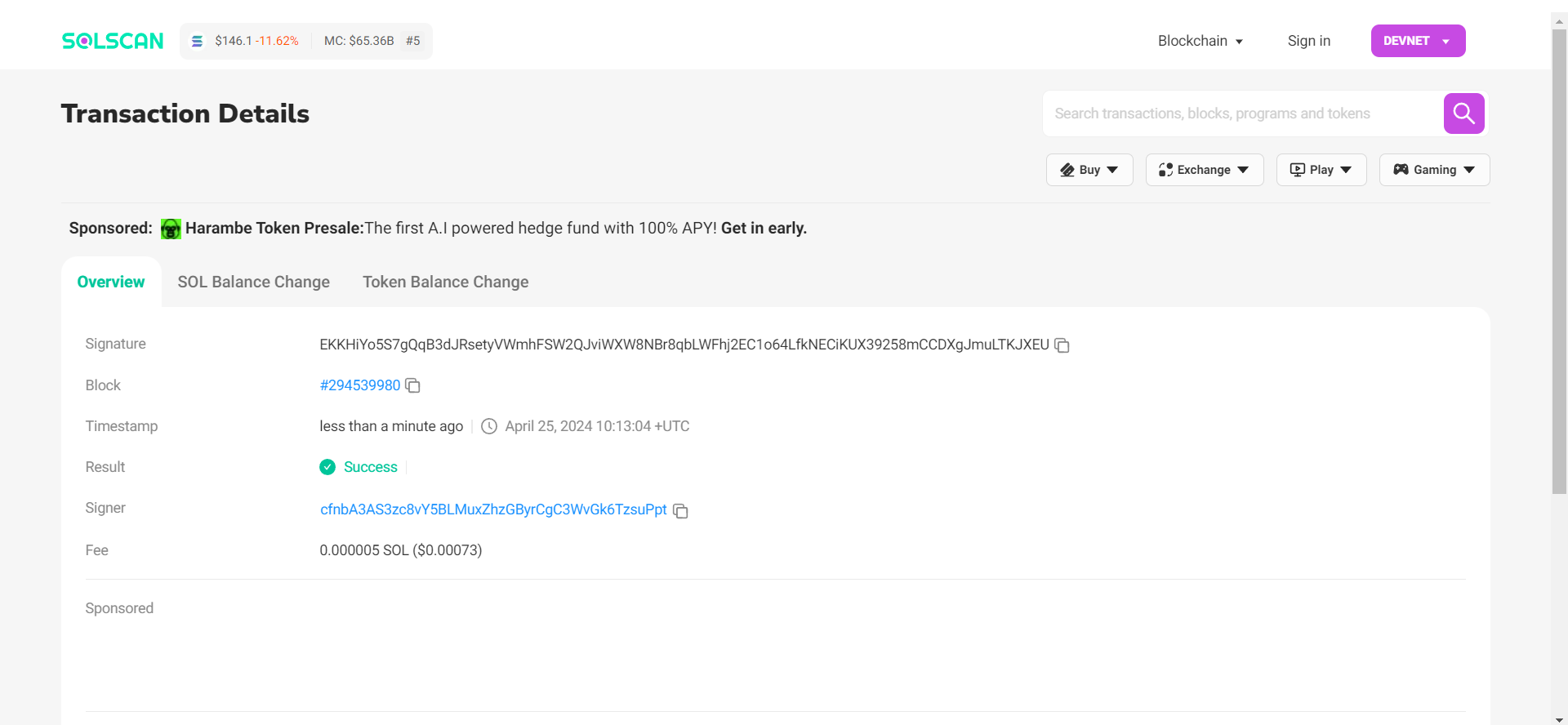
Note that we derive the PDA for each new account using the initializer’s public key and the movie title as optional seeds. Setting up the PDA this way restricts each user to only one review for any one movie title. However, it still allows the same user to review movies with different titles and different users to review movies with the same title.

Once we’ve calculated the rent and verified the PDA, we are ready to create our new account. In order to create a new account, we must call the create\_account instruction from the system program. We do this with a Cross Program Invocation (CPI) using the invoke\_signed function. We use invoke\_signed because we are creating the account using a PDA and need the Movie Review program to “sign” the instruction.

Now that we’ve created a new account, we are ready to update the data field of the new account using the format of the MovieAccountState struct from our state.rs file. We first deserialize the account data from pda\_account using try\_from\_slice\_unchecked, then set the values of each field.

We're ready to build and deploy our program!





**Basic Securityand Validation**

Let’s practice together with the Movie Review program we've worked on in previous lessons. No worries if you’re just jumping into this lesson without having done the previous lesson - it should be possible to follow along either way.

As a refresher, the Movie Review program lets users store movie reviews in PDA accounts. Last lesson, we finished implementing the basic functionality of adding a movie review. Now, we'll add some security checks to the functionality we've already created and add the ability to update a movie review in a secure manner.

Just as before, we'll be using Solana Playground to write, build, and deploy our code.

Let's delve into enhancing the security and validation aspects of our Movie Review program. In previous sessions, we established the foundation for users to store movie reviews in PDA accounts. Now, our aim is to bolster this functionality with security checks and the ability to update reviews securely.

1. Starter Code Overview: We've revamped the code structure by splitting it into three main files: lib.rs, entrypoint.rs, and processor.rs. This restructuring aims to improve readability and manageability. Additionally, we've introduced an error.rs file to define custom errors for better error handling.

2. Account Length Adjustment: To simplify the process and focus on security, we've allocated a fixed size of 1000 bytes to each review account. This eliminates the need for dynamic reallocation of account sizes.

3. Additional Functionality: We've enriched the MovieAccountState struct in state.rs with new functionality, including an is\_initialized function to verify if an account has been initialized.

4. Custom Errors: We'll define custom errors to handle specific scenarios such as uninitialized accounts, mismatched PDAs, oversized data, and invalid ratings. These errors will enhance the program's error handling capability.

5. Data Validation: We'll ensure that ratings provided by users fall within the acceptable range of 1 to 5. If a rating is outside this range, we'll trigger a custom InvalidRating error.

6. Support for Update: We'll extend the program to support updating movie reviews by adding an UpdateMovieReview variant to MovieInstruction. This will include data for the new title, rating, and description.

7. Implementing Update Logic: We'll implement the logic for updating movie reviews securely, iterating through the necessary accounts and ensuring robustness from the outset.

Once the implementation is complete, we'll proceed to build and test our program to ensure its functionality and security. Utilizing a frontend, we can submit transactions for testing, ensuring the program functions as expected.

This process reinforces the security and functionality of our Movie Review program, ensuring it meets the desired standards and provides a reliable user experience.