Challenge: Generic Storage with Multiple Serialization Formats

Objective

Build a generic storage system that can serialize and deserialize data using different formats (Borsh, Wincode and JSON). This challenge will help you understand Rust traits, generics, PhantomData, and how to work with multiple serialization libraries.

Requirements

1. Define a Serializer Trait

Create a `Serializer` trait with two methods:

- `to_bytes()` - converts data to bytes
- `from_bytes()` - converts bytes back to data

Both methods should:

- Be generic over the data type
- Return a `Result` type for error handling
- Work with any type that implements the necessary serialization traits

2. Implement Three Serializers

Create three structs that implement the `Serializer` trait:

- `Borsh` - using the borsh library
- `Wincode` - using the wincode library
- `Json` - using the serde_json library

Each implementation should handle serialization errors appropriately.

3. Create a Generic Storage Container

Build a `Storage<T, S>` struct that:

- Is generic over both the data type `T` and the serializer type `S`
- Stores serialized data as bytes internally
- Uses `PhantomData<T>` to maintain type information
- Has appropriate trait bounds on `T` (must be serializable by all three formats)

4. Implement Storage Methods

Your `Storage` should have at least:

- `new(serializer: S)` - create a new storage with a specific serializer

- `save(&mut self, value: &T)` - serialize and store data
- `load(&self)` - deserialize and return data
- `has_data(&self)` - check if data is stored

5. Create a Test Data Type

Define a struct (e.g., `Person` or `TestData`) that:

- Derives all necessary serialization traits
- Has at least 2 fields of different types
- Can be used to test all three serializers

6. Write Tests

Create unit tests that verify:

- Data can be saved and loaded with Borsh
- Data can be saved and loaded with Wincode
- Data can be saved and loaded with JSON
- Loaded data matches the original data

Learning Goals

By completing this challenge, you should understand:

- How to design and implement generic traits
- How to use PhantomData for zero-cost type tracking
- How trait bounds work with multiple serialization libraries
- The differences between various serialization formats
- Error handling with Result types
- How to write generic code that works with different implementations

Bonus Challenges (Optional)

If you want to extend the challenge:

1. Add a method to convert between different serializers
2. Add benchmarks to compare serialization performance

Expected Output

Your program should be able to run something like:

rust

```
let person = Person { name: "André".to_string(), age: 30 };

let mut borsh_storage = Storage::new(Borsh);
borsh_storage.save(&person).unwrap();
let loaded = borsh_storage.load().unwrap();
println!("Loaded: {:?}", loaded);
```

And successfully save/load data using all three serialization formats.