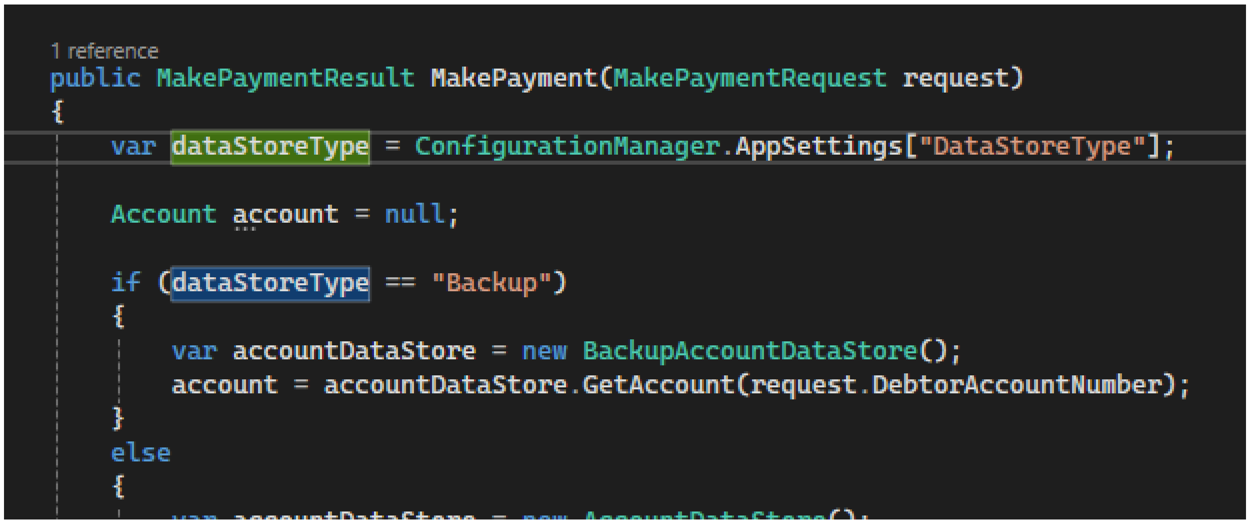
Technical test

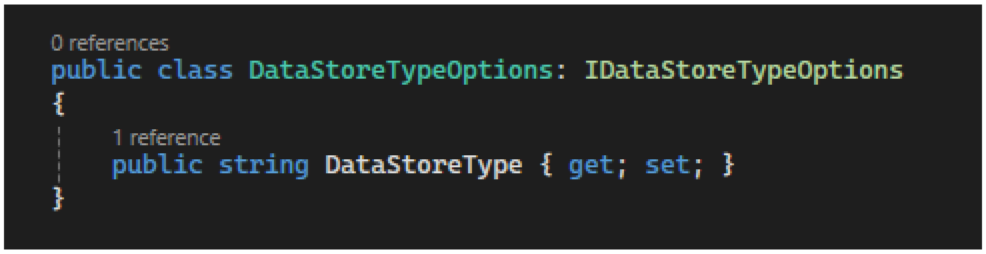
PaymentService.cs

The `dataStoreType` is a dependency here, and might be best to use strong typing



```
1 reference
public MakePaymentResult MakePayment(MakePaymentRequest request)
{
    var dataStoreType = ConfigurationManager.AppSettings["DataStoreType"];

    Account account = null;

    if (dataStoreType == "Backup")
    {
        var accountDataStore = new BackupAccountDataStore();
        account = accountDataStore.GetAccount(request.DebtorAccountNumber);
    }
    else
    {
```

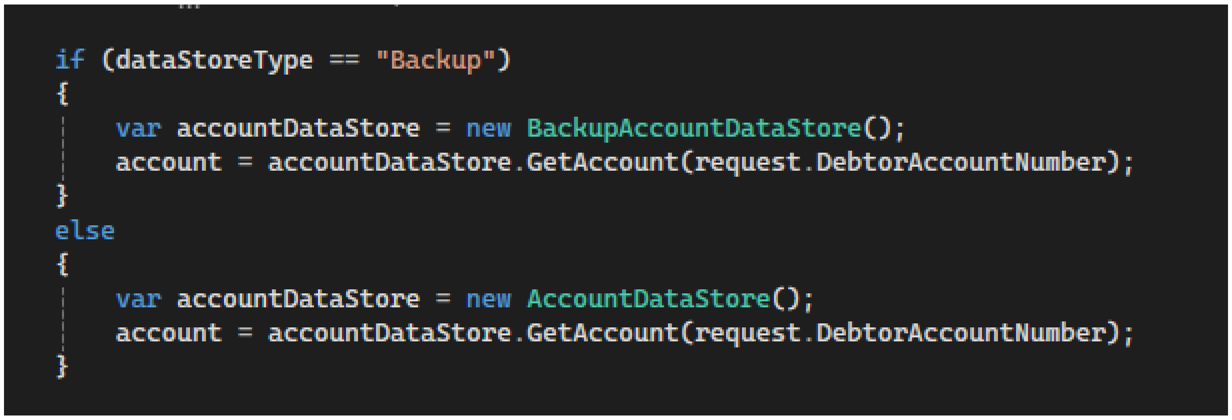
The value retrieved for it is a string, might be better as a typed value.
I created a class and corresponding interface for this



```
0 references
public class DataStoreTypeOptions: IDataStoreTypeOptions
{
    1 reference
    public string DataStoreType { get; set; }
}
```

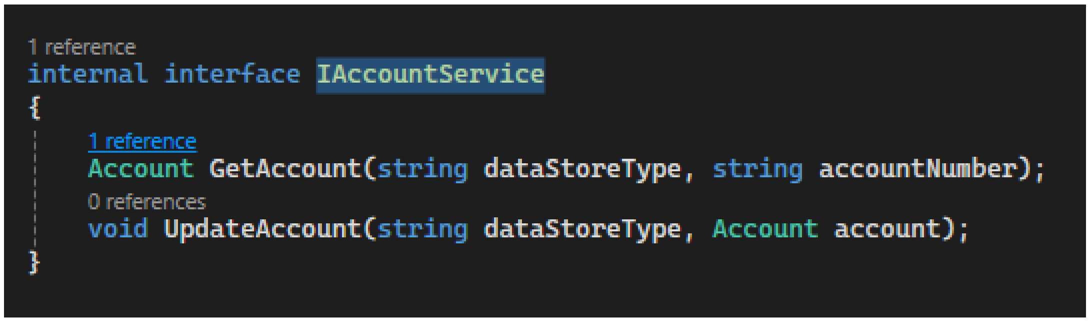
In `Program.cs` this would be bound using `config.GetSection()` and then `.AddSingleton()` for DI (or another DI scope)

Here, we’re creating two different data stores

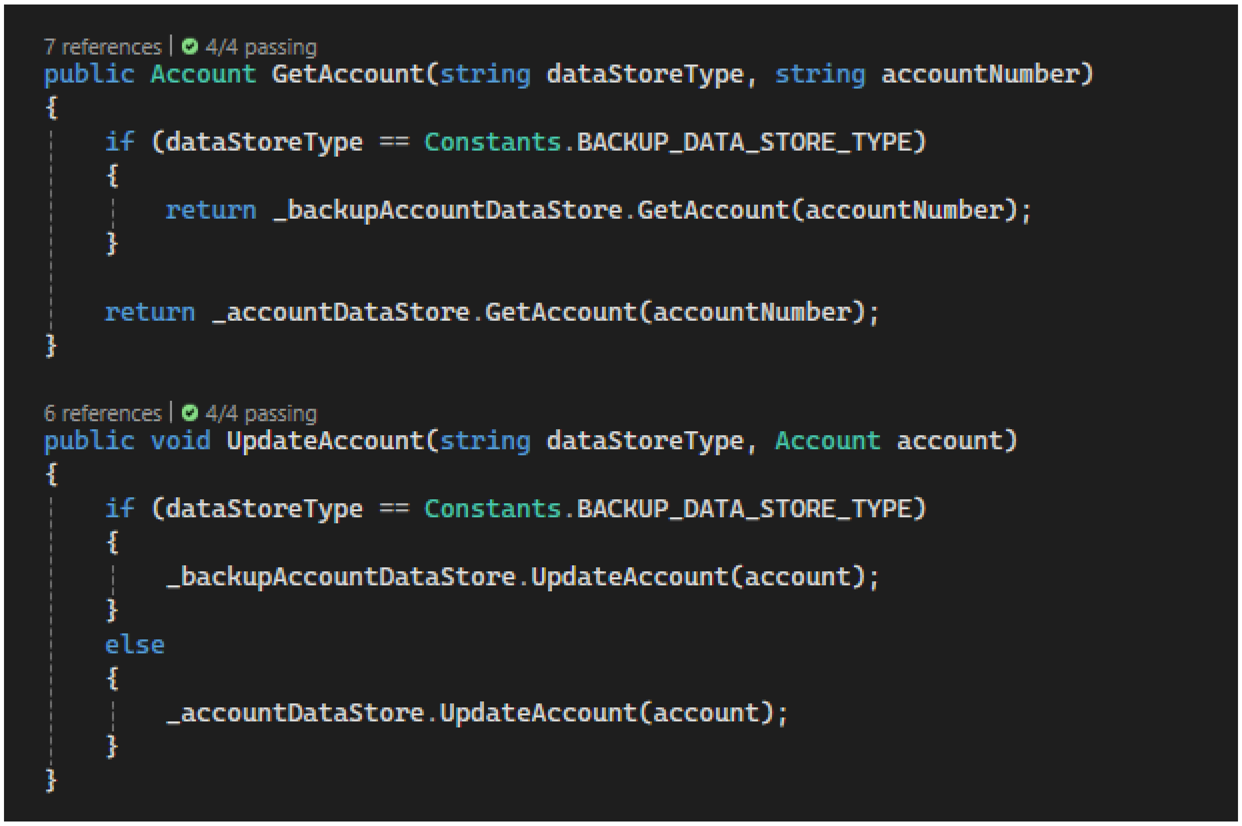

```
if (dataStoreType == "Backup")
{
    var accountDataStore = new BackupAccountDataStore();
    account = accountDataStore.GetAccount(request.DebtorAccountNumber);
}
else
{
    var accountDataStore = new AccountDataStore();
    account = accountDataStore.GetAccount(request.DebtorAccountNumber);
}
```

Both have the same functions - they may have different implementations, so I created an interface for them and moved them into their own service, `IAccountService` and `AccountService`

The service will implement this interface

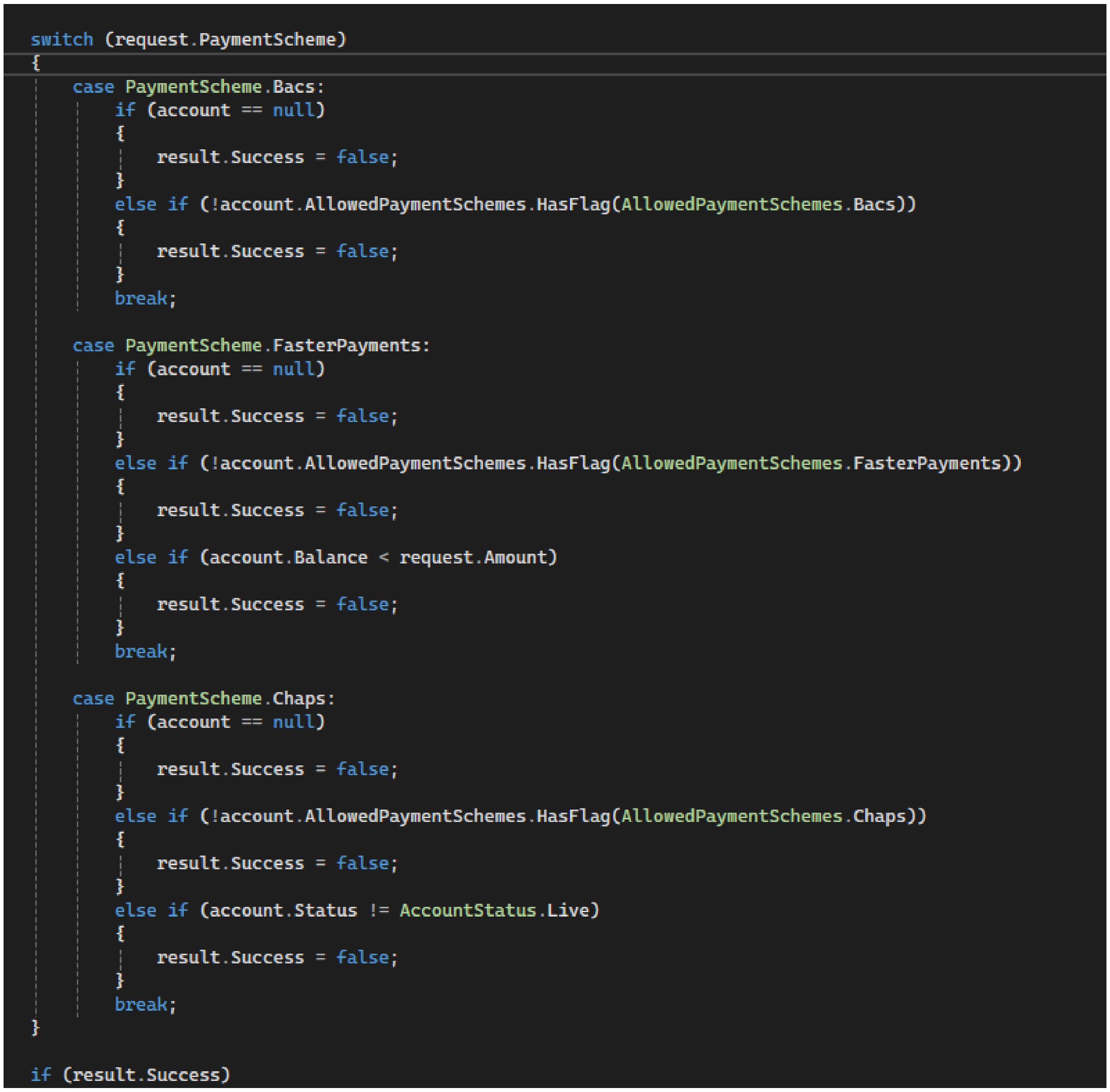

```
1 reference
internal interface IAccountService
{
    1 reference
    Account GetAccount(string dataStoreType, string accountNumber);
    0 references
    void UpdateAccount(string dataStoreType, Account account);
}
```

This `AccountService` will end up returning the account from the relevant data store, and updating the account using the relevant data store



```
7 references | 4/4 passing
public Account GetAccount(string dataStoreType, string accountNumber)
{
    if (dataStoreType == Constants.BACKUP_DATA_STORE_TYPE)
    {
        return _backupAccountDataStore.GetAccount(accountNumber);
    }

    return _accountDataStore.GetAccount(accountNumber);
}

6 references | 4/4 passing
public void UpdateAccount(string dataStoreType, Account account)
{
    if (dataStoreType == Constants.BACKUP_DATA_STORE_TYPE)
    {
        _backupAccountDataStore.UpdateAccount(account);
    }
    else
    {
        _accountDataStore.UpdateAccount(account);
    }
}
```

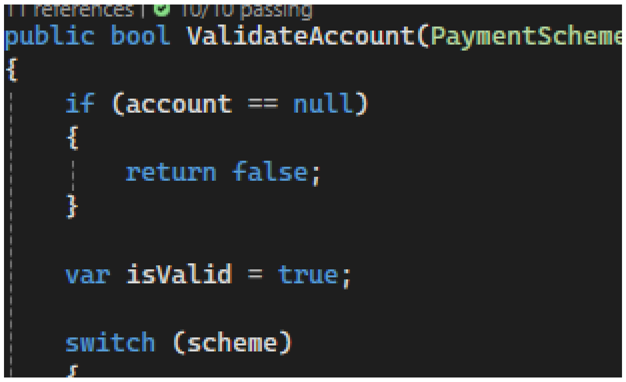
This middle section is another piece of logic that can be refactored into another service



```
switch (request.PaymentScheme)
{
    case PaymentScheme.Bacs:
        if (account == null)
        {
            result.Success = false;
        }
        else if (!account.AllowedPaymentSchemes.HasFlag(AllowedPaymentSchemes.Bacs))
        {
            result.Success = false;
        }
        break;

    case PaymentScheme.FasterPayments:
        if (account == null)
        {
            result.Success = false;
        }
        else if (!account.AllowedPaymentSchemes.HasFlag(AllowedPaymentSchemes.FasterPayments))
        {
            result.Success = false;
        }
        else if (account.Balance < request.Amount)
        {
            result.Success = false;
        }
        break;

    case PaymentScheme.Chaps:
        if (account == null)
        {
            result.Success = false;
        }
        else if (!account.AllowedPaymentSchemes.HasFlag(AllowedPaymentSchemes.Chaps))
        {
            result.Success = false;
        }
        else if (account.Status != AccountStatus.Live)
        {
            result.Success = false;
        }
        break;
}

if (result.Success)
```

Once moved into a new service, `AccountValidationService`, I then wrote the tests for it. Firstly commenting out everything and testing each block individually until all tests passed.

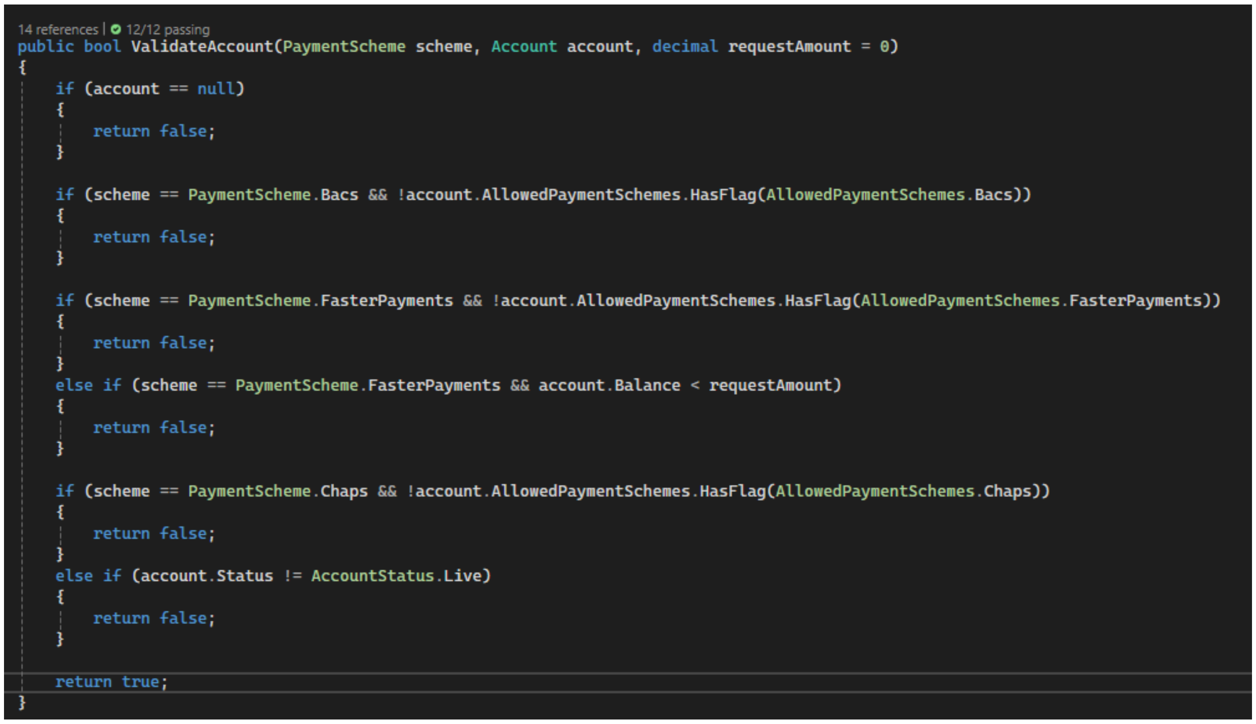
Next, there was a lot of repeated code, so the class itself could be simplified. If the account is null in all cases, return `false`, so this was moved to the top of the class as an early return.



```
11 references | ✓ 10/10 passing
public bool ValidateAccount(PaymentScheme
{
    if (account == null)
    {
        return false;
    }

    var isValid = true;

    switch (scheme)
```

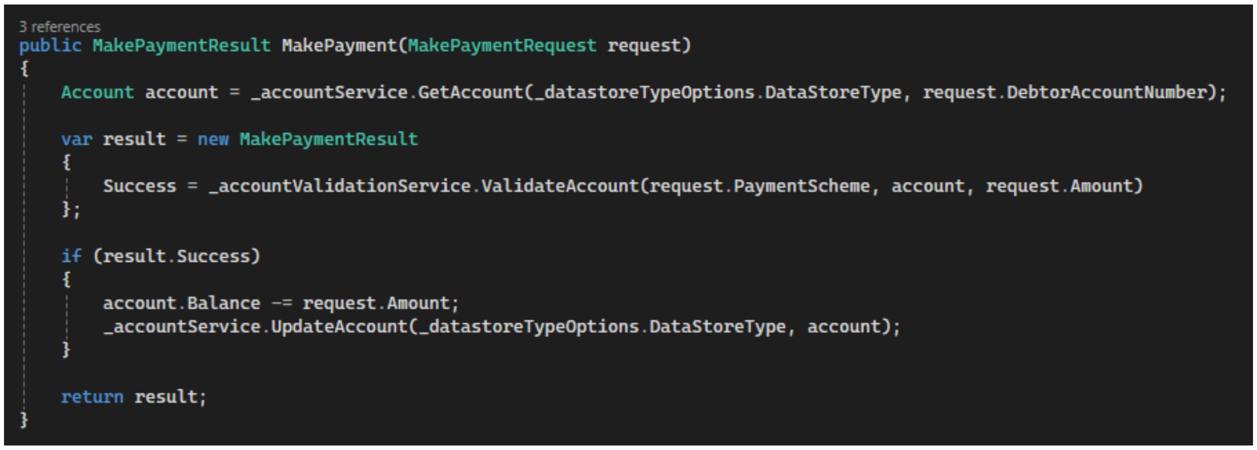
I decided to leave the separation between `AllowedPaymentSchemes` and `PaymentScheme` alone as I felt it may introduce bugs by removing one over the other, given that `AllowedPaymentSchemes` uses bitshift operators to give specific values. If it were to unify it, I would make `account.AllowedPaymentSchemes` an array of `PaymentScheme`, and check whether the array contains the given scheme passed into the `ValidateAccount()` method.

I did refactor the `switch` statement to multiple if-statements that returned early. I felt that having a case statement with nested if-blocks adds cognitive load, having to keep track of what conditions were used to get to the given block.I achieved this by testing the original case statement, and then refactoring to do early returns. I feel that this is much easier to read than a case statement with nested if-statements.



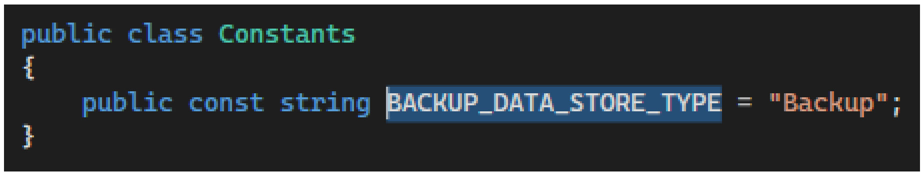
```
14 references | ✓ 12/12 passing
public bool ValidateAccount(PaymentScheme scheme, Account account, decimal requestAmount = 0)
{
    if (account == null)
    {
        return false;
    }

    if (scheme == PaymentScheme.Bacs && !account.AllowedPaymentSchemes.HasFlag(AllowedPaymentSchemes.Bacs))
    {
        return false;
    }

    if (scheme == PaymentScheme.FasterPayments && !account.AllowedPaymentSchemes.HasFlag(AllowedPaymentSchemes.FasterPayments))
    {
        return false;
    }
    else if (scheme == PaymentScheme.FasterPayments && account.Balance < requestAmount)
    {
        return false;
    }

    if (scheme == PaymentScheme.Chaps && !account.AllowedPaymentSchemes.HasFlag(AllowedPaymentSchemes.Chaps))
    {
        return false;
    }
    else if (account.Status != AccountStatus.Live)
    {
        return false;
    }

    return true;
}
```

Calling this service from `PaymentSevice` makes it far easier to read what the payment service class is now doing



```
3 references
public MakePaymentResult MakePayment(MakePaymentRequest request)
{
    Account account = _accountService.GetAccount(_datastoreTypeOptions.DataStoreType, request.DebtorAccountNumber);

    var result = new MakePaymentResult
    {
        Success = _accountValidationService.ValidateAccount(request.PaymentScheme, account, request.Amount)
    };

    if (result.Success)
    {
        account.Balance -= request.Amount;
        _accountService.UpdateAccount(_datastoreTypeOptions.DataStoreType, account);
    }

    return result;
}
```

Finally, I set the “Backup” string as a constant to avoid mistyping



```
public class Constants
{
    public const string BACKUP_DATA_STORE_TYPE = "Backup";
}
```

Notes on AccountDataStore and BackupAccountDataStore

The `AccountDataStore` and `BackupAccountDataStore` both implement the same functions, but I needed a way to distinguish between both of them as both were needed in the `AccountService` implementation. Therefore the common methods were moved into a single `IDataStore` interface, and I used an empty interface that implements `IDataStore` to be able to distinguish between both, and these can now be passed in as separate interfaces to `AccountService`. I know that an `IEnumerable<IDataStore>` could have been passed in instead, but I would then have had to include logic to find the correct one using reflection, rather than passing both in and simply calling the correct one

If code was to be implemented, then the tests would follow this logic

- `GetAccount`
  - If the account doesn’t exist, then return null, as the `ValidateAccount` function in the `AccountValidationService` checks for its existence
  - If it does exist, then return the account
- `UpdateAccount`
  - Change the return type to be a `bool`

- If the account doesn’t exist, then either throw an exception to say it’s not found, or return `false` (this is preferred) and return that from `MakePayment()`, because if the function was left as `void`, the `MakePaymentResult` would still be `true` for a failed update
- If the account exists and can be updated, return `true` from it, and then set the `Success` property on the result object on `MakePaymentResult`