# Scenario 01 – Migrated custom application

This scenario is built to show customer that there are tools available to find resolutions to error which at first look in the logs don’t really give you enough details. The code contains a webpart that relies on configuration data stored in the configuration database as a SPPersistedObject. After migrating sites to another farm that has this webpart provisioned, that configuration object is not available and a NullReferenceException is thrown. By using a disassembler (e.g. ILSpy, .NET Reflector etc) the error can be tracked down and resolved without having access to the source code.

## Origin

This scenario is originally inspired from helping a fellow PFE troubleshoot the tooling provided as part of the RKM for SharePoint delivery. This is installed as a farm solution in a SharePoint 2010 environment. After backing up the site collection and restoring it to another farm the configuration data was missing because a farm scoped feature had not been activated in destination farm. Using ULS logs to find the call stack and extracting the DLL from the solution package the code could be disassembled and the origin of the problem could be found, and resolved in a matter of minutes.

## Prerequisite Lessons

For the students to work with this scenario they should have been through the **ULS** and **Disassembler** lessons.

## Tasks

By leveraging demonstrated troubleshooting tools, you should complete the following steps as part of this troubleshooting scenario:

1. Find where in the application the exception occurs. To answer this question you should provide the following:
   1. Assembly name
   2. Class
   3. Member
2. Find a resolution to the error in the target environment
3. Go beyond the error itself and explain what the root cause is?
4. What options could the developers have considered in order to avoid the error?

## Answers and key discussion points

This section highlights some of the key discussion points of the scenario

### Answers

1. Below are the answers to the three parts:
   1. Scenario01.WebParts.dll
   2. FeedWebPartUserControl
   3. Ctor
2. Activate the farm feature called “Scenario 01: Global Config”
3. The root cause is that the configuration dependency does not migrate seamlessly with the webpart component in the application. This is a design decision that developers/architects should consider when choosing configuration model.
4. There are several ways the developer could have considered to avoid this:
   1. Storing configuration in a list
   2. Storing configuration as a property bag entry
   3. Storing configuration as a web part property
   4. Set the farm feature to auto-activate

### Configuration data

There is not a single answer to how a developer should implement storage of configuration data. It will vary depending on the type of data, and at which scope the data should be configurable.

If the component using the data is locally contained in a site or site collection the decision falls naturally on using either a list to store it, or as a property bag entry. The benefit of storing in a list is that is has automatically gets built-in editing support through list forms. A property bag entry is quicker to implement though, and in case site admins should not be able to configure through UI this works fine.

If the configuration is local to a page a web part property would be a more natural choice. Although the ability for reuse on more limited.

If the scope of the configuration data is higher than site or site collection there isn’t really any good choices that work seamlessly in a backup/restore scenario. Storing it in web.config has the drawbacks of application pool recycling in the event of changing the configuration. Storing data in SPPersistedObject becomes farm scoped, and is easier to implement compared to web.config entries. Although editing capabilities are limited unless the developer provides additional tooling for editing the data stored in the configuration database.

Discussion around this topic should highlight the importance of developers and architects to consider the long term contextual changes that can happen to the hosting environment where their customizations are deployed and executed. Disaster recovery situations, and how customizations behave in this scenario, is especially important to consider. In the event that disaster recovery routines have to be invoked, customizations can add additional complexity to an already complex scenario that comes with SharePoint. Lowering this complexity and thereby the dependency on knowledge that only developers have, should be part of the mindset when architecting any custom solution on top of SharePoint.