# Progress Spark Toolkit Component Guide

Available components within the Progress Spark Toolkit

August 2018

# Summary

The **Progres Spark Toolkit** is a reference implementation of the **Common Component Specification** (CCS) framework, which was released as version 1 and open-sourced. By default the CCS consists primarily of class interfaces and no functional code, thus the need to illustrate how to build a sample application which utilizes these interfaces. The included components are meant to provide features which can be considered “standard” for most web-based applications intended to operate in a stateless environment.

# Prerequisites

To provide the best experience with the **Progress Application Server (PAS)** and **Progress Developer Studio (PDSOE)** it is recommended that you be on the latest service pack of OpenEdge. The source provided with Spark should be compatible with both OpenEdge 11.6 and 11.7, with the latter being preferable due to significant simplifications of security options and improved support for Single Sign-On and OAuth. Use of a **64-bit Windows** installation and **OE 11.7.2 or later** is assumed, and at least the Progress Developer Studio for OE component present.

Access to the repository is provided through **Git** and you may benefit from having a suitable Git client installed on your workstation. As a suggestion, **Git for Windows** and **TortoiseGit** will provide a seamless integration with Window Explorer. Some automated tasks will be performed using **Ant** which will already be present in your DLC directory if using 11.7 or later.

# Obtaining Code

1. Create a directory **C:\Modernization** for all future application code.
2. **Option 1:** Right-click within the new directory to view TortoiseGit options.
   1. Perform a “**Git Clone**” of [**https://github.com/progress/Spark-Toolkit**](https://github.com/progress/Spark-Toolkit)
3. **Option 2:** Visit <https://github.com/progress/Spark-Toolkit/releases> and download the latest available release as either a .ZIP or .TAR.GZ archive (~90MB).
   1. Expand the archive, making sure the top-level directory is named simply “**Spark-Toolkit**” and contains an immediate “**src**” folder within.
4. Confirm the source code is available by locating the “**src**” directory contents.
   1. This directory can be imported into PDSOE as an OpenEdge project.

# Workspace Options

Before proceeding, it may be useful to enable some options within the Progress Developer Studio for OpenEdge (Eclipse) environment. These options will provide a more consistent experience with the actions to be requested in the remainder of this document. Begin by starting the **Developer Studio** and selecting **C:\Modernization** as your workspace location. If PDSOE has already been started under a different workspace, use the option **File > Switch Workspace > Other…** to choose.

Window -> Preferences

General

Show heap status: checked

Editors

Text Editors

Insert spaces for tabs: checked

Displayed Tab Width: 4

Show line numbers: checked

Search

Reuse editors: unchecked

Workspace

Refresh using native hooks or polling: checked

Refresh on access: checked

Workspace name: "Your Workspace Name Here"

LocalHistory

Maximum entries per file: 1

Progress OpenEdge

Editor

Case: Lower

Expand keywords: checked

Case keywords: checked

Build -> Automatically syntax check: checked

Server

Remove all files and folders when cleaning server pub dir: checked

Update properties from server before starting/launching: checked



Project Explorer -> View Menu (small icon on panel, shown above)

Customize View

Select the filters to apply (matching items will be hidden)

Unselect \*.pl to view Procedure Library files.

# Common Questions

**Q: Why would I want to use the Progress Spark Toolkit?**

A: Building a modernized application can consist of both many moving parts as well as best-practices that may only be apparent after consuming all of the relevant product documentation. The provided features in the **Progress Spark Toolkit** are build specifically to address the needs of a secure business application, while providing a known path for building such a solution. In addition, the use of the CCS library as a base for all features means you’re not locked into a single, rigid solution from any one vendor.

**Q: Couldn’t I build something on my own?**

A: Absolutely. Neither the presence nor absence of the **Progress Spark Toolkit** should preclude you from implementing your own solution. What the toolkit aims to do is alleviate any significant investment of time and money to build, test, implement, and improve a home-grown solution for building modernized web applications.

**Q: How do I get started with the Progress Spark Toolkit?**

A: If you are just getting started with the **Progress Spark Toolkit** please refer to either the “Spark Evaluation Guide” to build one of the demo projects provided, or follow the “Spark Quick-Start Guide” to begin your own custom project. Otherwise, if you may be inheriting an existing project and need more information about the available components, continue reading this guide for more explanation and in-depth code samples.

**Q: I don’t want the default behavior, what should I do?**

A: There are several options for overriding behavior available to you when you implement the components of the **Progress Spark Toolkit**.

First, many of the provided managers may be simply turned off if they are not relevant to your needs. This will be addressed later in this document.

Second, you may make use of PROPATH rules to provide your own copy of a manager—**however**, this may pose potential problems if attempting to upgrade the toolkit at a later time. This can offer a quick fix but is strongly discouraged.

Third, any manager provided within the toolkit may be overridden with your own custom implementation—either by inheriting and overloading the original class, or completely replacing the original with your own implementation. This will be covered in more detail later in this document.

# Manager Classes

Required Managers

The CCS v1 dictates that 3 core managers be implemented as part of the standard application stack: **StartupManager**, **ServiceManager**, and **SessionManager**. These may be referred to as the “Triple-S” managers, providing the following functionality:

**StartupManager** – This is the primary class that drives all other managers. It’s a class accessed via its own static **Instance** property, which creates the class instance if it does not already exist. It is during this instantiation process that various configuration files are read from disk and can alter the behavior of the class and its descendants. This requires a **startup.json** config file to operate.

**ServiceManager** – Creates a service implementation and can execute said service for a given lifecycle. This requires a **service.json** config file to operate. This is an open-ended and generic class.

**SessionManager** – Creates and manages user context within the application, after asserting the identity of the user against the connected database(s). Also provides methods to set or reset context attributes within a **ClientContext** object instance. This requires a **session.json** config file to operate.

Related to the 3 core managers above is a special class which acts as a “binding agent” to glue these pieces together. Accessed as **Ccs.Common.Application**, this class has no functional code but rather provides static properties to act as pointers to instances of the 3 classes above. The exact usage of this class will be addressed later in this material.

Suggested Managers

These are managers which can provide common behavior for your application:

**ConnectionManager** – Creates an external connection to any service as defined in the configuration. A typical use is for making and pooling additional AppServer connections.

**LoggingManager** – Fairly self-explanatory, provides logging capabilities within the framework. It is also used to capture errors and handle certain types in a specific manner.

**StateManager** – Utilized by the session manager to read/write session data. When a session is started the **SessionManager** creates the initial object, and this class stores the data (default: flat file) when the session is ended. When re-establishing a session, the **SessionManager** reads the existing context and populates the context object. This class would typically be overridden, for example, to provide that context storage within a database.

**StatsManager** – Tracks common request/response information for reporting and statistics purposes, though by default will provide some level of debugging when the agent logging-level is set to 3 or higher.

**TranslationManager** – Provides a single point of override for translating text. This is an open-ended and generic class.

Optional Managers

If following the standard pattern for business entities as generated by the Progress Developer Studio, then all artifact mapping and code execution comes from use of generated artifacts and potentially the **OpenEdge.Web.DataObject.DataObjectHandler** class. However, there are some highly-specific managers that may only be useful or necessary if you intend to follow a pattern which dynamically discovers and executes business logic which inherits the Spark **DynamicResource** or **DynamicEntity** classes:

**CatalogManager** – Provides a means of automatically registering ABL classes or procedures as REST resources, and producing a catalog structure as required by the Progress JSDO (by default).

**SchemaManager** – Utilized (read: required) by the **CatalogManager** to dynamically register schema information either from a connected database or an included dataset or temp-table definition.

**RouteManager** – Called by the legacy service interface, and determines how to execute the request. Typically creates the necessary service implementation via the ServiceManager, first. ***This may be deprecated in a future release.***

**MessageManager** – In toolkit terms a message is an instance of a request or response object. This manager provides handlers for certain built-in types that handle more complex operations than the standard JSON request/response. This is separate from the standard WebRequest or WebResponse objects used by a WebHandler class. However, access of the request/response within ABL code breaks the ability to place test facades and other mock interfaces in front of your application code. ***Therefore, this feature may be deprecated in a future release.*** In place of utilizing the MessageManager, it is encouraged to declare all of your input/output parameters in your exposed ABL logic and use the mapping features of your Data Object Service (through the DataObjectHandler pattern) to assign these directly to the necessary HTTP artifact (header, body, etc.).

# Startup/Shutdown

The process of including the **Progress Spark Toolkit** is as simple as kickstarting the **Ccs.Common.Application** class with an instance of a **StartupManager** implementation. This is accomplished through the default **Spark/startup.p** procedure. Whether the provided startup procedure is used as-is or run from another custom startup procedure, so long as the following code is included for execution the Spark application stack will be started:s

Ccs.Common.Application:StartupManager =

Spark.Core.Manager.StartupManager:Instance.

As part of the process for accessing the static **Instance** property of the **StartupManager**, if a valid instance does not already exist then a new one will be created. As part of this process, the **ServiceManager** and **SessionManager** will also be started automatically and set to their respective static properties of the **Application** class. Finally, as part of the ServiceManager’s initialization process any of the suggested and optional managers configured and available will also be started.

Regarding the **Spark/startup.p** procedure, in addition to starting the “Triple-S” managers the provided procedure also starts a custom handler class to manage events from the **OpenEdge.Web.DataObject.DataObjectHandler**, when utilized and if running OE 11.6.3 or later. This is enabled in the startup code by simply creating a new instance of the following class:

new Spark.Core.Handler.DOHEventHandler().

To ensure that managers are stopped and objects removed from agent memory, a default shutdown procedure **Spark/shutdown.p** is provided to perform this action. In addition, when the **log-manager:logging-level** is set to 3 or higher the script will automatically output a list of objects, buffers, handles, queries, sockets, and procedures still in memory. This can be a helpful tool for identifying memory leaks within your application. As with the startup procedure you are free to create your own code, and can simply run the original procedure to perform default actions if desired.

# Directory Structure

All toolkit code is contained within a “**Spark**” folder, with the immediate components located within a “**Core**” folder. As a point of history, originally there was a separate “UI” folder which added a presentation layer above the Core components, but was removed as it no longer fit within the strategy of the Spark efforts. To avoid breaking legacy code and projects implemented so far using the framework, this core folder was retained as part of the class path.

**Constant** – Static values and constants

**Handler** – Implementations of WebHandler classes

**Interface** – Service interfaces for the RouteManager and façade classes

**Lib** – Include files for common features

**Manager** – Manager implementations and customized class interfaces

**Message** – Message classes for the MessageManager

**Security** – User and data security modules, eg. OERealm and hashing

**Service** – Service classes for the ServiceManager

**Util** – Utility classes and common application tools

**Web** – Extensions of the OpenEdge WebRequest and WebResponse classes

When building your own application and the need arises to override a class, it is encouraged that you create your own namespace for your class but retain the same directory structure for the override class. For instance, if you planned to override the **Spark.Core.Manager.SessionManager** class the resulting namespace might be **MyProject.Spark.Core.Manager.SessionManager** (replacing with an appropriate project namespace). This maintains a familiar structure and intended purpose for the enclosed files. As another example, if you wanted to create your own set of project utilities the resulting path may look like **MyProject.Util.ClassName**.

# Configuration Files

Each manager listed above has a corresponding JSON configuration file. In most cases the configuration file MUST be present in order for the manager to correctly load, while in some rare cases the absence of a configuration means default options will be used. To ensure transparency in operation a sample configuration for critical managers should always be present with the available toolkit code you downloaded.

The naming scheme for the files is straightforward: each config should directly relate to the manager it runs. For instance, the StartupManager should have a startup.json present. By default the config files will be located using the following directories, in order, where CATALINA\_BASE is your PAS instance location:

1. <CATALINA\_BASE>/conf/<CONFIG\_PROJECT\_DIR>
2. OS Environment Variable “SPARK\_CONF”
3. <CATALINA\_BASE>/conf/spark
4. <CATALINA\_BASE>/conf

For the path option “CONFIG\_PROJECT\_DIR”, this may be set as a session startup procedure parameter (sessionStartupProcParam), stated as a JSON string. This allows you to provide a project-specific configuration directory as part of the startup for your ABLApp—something extremely useful if utilizing multiple ABLApps within a single PAS instance. For an example, see the following option to set the configuration directory to <CATALINA\_BASE>/conf/sports:

sessionStartupProcParam={"ConfigDir": "sports"}

In most cases the structure of these files represents a ProDataSet (multi-table) or Temp-Table (single) in their JSON form. As such, each file **must** be valid JSON to be parsed correctly, and use of tools such as JSONLint (<https://jsonlint.com/>) is highly suggested when editing the files by hand. For the required “Triple-S” managers this is obvious as the outermost object (dataset) is named “Config” and contains 1-2 child objects (tables) for configuring various aspects of the manager.

# Customizations

Now that we’ve introduced the configuration files, we can use these to extend behavior by creating customized classes. We’ll begin by examining one of the configuration files and what we can do to modify it for more specific needs.

Configuration Changes

The first configuration file you may encounter is **startup.json** for the StartupManager class. As previously noted, the first node of the file will be a “**Config**” object which should contain a “**ManagerMapping**” array. Each object within this array should define a “**Manager**” interface and the name of a class “**Implementation**”. Each manager name should be a standard Spark interface or class which inherits the appropriate CCS manager interface and any implementation should likewise implement the stated interface to be considered a valid class.

To examine this process further, as part of the StartupManager’s initialization, it checks for any managers listed by the config file. As an example, the **SessionManager** class for Spark is set by default to the **Spark.Core.Manager.IServiceManager** interface and means any valid class implementation should “implement” this interface. In our example case, the default implementation is **Spark.Core.Manager.ServiceManager** which implements the interface as necessary, and in turn inherits the following structure:

* Spark.Core.Manager.ServiceManager
  + Spark.Core.Manager.IServiceManager
    - Ccs.Common.IServiceManager
      * Ccs.Common.IManager
        + Ccs.Common.IService

This allows the framework to discover certain aspects about the manager in question. For example, if we used the IsA() method in ABL to check if the implementation were a “Ccs.Common.IServiceManager” instance, the answer would be yes as it inherits the appropriate class and therefore should contain any methods dictated by the interface implemented (eg. getService and stopServices). From this set of checks and balances we gain the power to replace default implementations with custom classes, as the interface demands that any class contain certain standard methods.

Extending Classes

The best part about the inheritance noted above is that it’s still possible to extend a class interface while maintaining backwards compatibility with the toolkit defaults. Let’s look at a common example in another configuration file, **service.json** for the **ServiceManager** class. Similar to the previous file, we should have a main “**Config**” object and a “**ServiceMapping**” array. Just like the startup class/config we have a “**Service**” interface class name and an “**Implementation**” class name. By default the **ClientContext** object that will be used can track some basic information about the user’s identity, though it may be common to set up some additional properties about a user when they first use the toolkit after logging in. To do this, we will want to extend the default class and provide our own custom overrides.

To begin, we need to create a new class which inherits the original. To illustrate this, refer to the **Sports.Spark.Core.Manager.ClientContext** included within the **Sports** demo project. Naturally, this begins by inheriting the original user context class **Spark.Core.Manager.ClientContext** and **Spark.Core.Manager.IClientContext** interface. From a functional standpoint you can now override any methods necessary from within the new class, though the next trick is to tell the application to use this alternative class. This is easily accomplished by changing the “Implementation” property of the appropriate **ServiceMapping** config object. In our case here this will be the new “Sports.Spark.CoreMmanager.ClientContext” class name.

Now there is a small caveat with all of this—we may need to cast our final instances to our custom datatype before we can effectively use it. Why? Quite simply, if we can override methods then we can add them as well. That means we have a new and improved class, but references to the class are passed as the original interface. Here is a quick example of how we could access our new class and ensure that we can still access any new methods/properties added:

define variable oClientContext

as **Sports.Spark.Core.Manager.ClientContext** no-undo.

oClientContext = cast(Ccs.Common.Application:SessionManager

:CurrentClientContext,

**Sports.Spark.Core.Manager.ClientContext**).

So why is this necessary? Within the **Ccs.Common.Application** class, the **SessionManager** property points to the current SessionManager instance. However, the **CurrentClientContext** object is of type **Spark.Core.Manager.IClientContext** and only whatever methods it exposes. In order for the compiler and our development environment to be aware of any changes we made, we must cast this object instance to that of our customized class. This same pattern can be applied to any other classes that you wish to override, and can be applied to any custom interfaces as well.