# Spark Component Guide

Available components within the Spark toolkit

March 2018

# Summary

The **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 this code. These components are meant to provide features that can be considered “standard” for most web-based applications which 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-Server**](https://github.com/progress/Spark-Server)
3. **Option 2:** Visit <https://github.com/progress/Spark-Server/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-Server**” 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.

# Manager Classes

Required Managers

The Common Component Specification dictates that 3 core managers be implemented as part of a typical application stack: **StartupManager**, **ServiceManager**, and **SessionManager**. Here are the “Triple-S” managers and what they provide:

**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.

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 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.

**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).

**RouteManager** – Called by the service interface, and determines how to execute the request. Typically creates the necessary service implementation via the ServiceManager, first.

**MessageManager** – In Spark 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.

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

# Startup/Shutdown

The process of including the Spark toolkit is as simple as kickstarting the **Ccs.Common.Application** class with an instance of a **StartupManager** implementation. This is accomplished through the **Spark/startup.p** procedure. In addition to starting the “Triple-S” managers and any supporting services, a custom handler class is started to manage events from the DataObjectHandler, when utilized.

To ensure that managers are stopped and objects removed from agent memory, a shutdown procedure **Spark/shutdown.p** exists 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.

# Directory Structure

All toolkit code is contained within a “**Spark**” folder, with the immediate components located within a “**Core**” folder. To note, originally there was a separate “UI” folder which added a 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 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