PostSharp. OnFieldAccess

Прочитать статью на сайте

И снова речь пойдет о PostSharp! =) Сегодня я хочу рассказать об аспекте **OnFieldAccess** (или в версии 2.0 **LocationInterceptionAspect**), который позволяет держать контроль над использованием свойств и полей класса. В прошлых статьях я рассказывал о применении аспектов на вызовы методов и о задачах, которые могут быть решены с их помощью. В целом, все те же задачи применимы и к полям: логирование изменения значений полей, защита доступа по ролям. Кроме этого можно организовать защиту полей от записи, декларативную проверку значений, уведомление других объектов об изменении значений.

Многие эти задачи при реализации традиционным способом, порождают необходимость написания кучи однородного кода. PostSharp позволяет избавиться от размазывания кода по всем слоям и консолидирует все проверки в одном месте, что, на мой взгляд, легче для понимания и дальнейшего внесения изменений.

Так же, в этой статье я рассмотрю такую возможность PostSharp, как проверка возможности применения аспекта на этапе компиляции. Я считаю, что это замечательная возможность, которой не стоит пренебрегать в дальнейшем. Для кого-то это может даже стать решающим моментом в применении данного продукта. Я знаю таких людей, которые ни за что не будут пользоваться продуктом, который не будет их строго ограничивать, а тут они имеют возможность написать себе такие ограничения и радоваться жизни. Да и в случае дистрибуции третьим сторонам вашей библиотеки это будет полезно.

Для опробования дальнейших примеров вам необходимо скачать и установить **PostSharp 2.0**.

Security

Как уже было сказано выше, с помощью PostSharp можно настроить доступ к полям классов по ролям и, например, разделить права на запись и чтение. Допустим, у нас есть какой-то класс и две роли: администратора и пользователя. Необходимо сделать так, чтобы определенные свойства не были доступны на запись пользователю.

Итак, пусть это будет классом, который нуждается в защите:

```
public class SecurableClass {
        public string SharedField { get; set; }
        public string SecureField { get; set; }
    }
```

Как видно из названия, необходимо обеспечить выборочный доступ для поля SecureField. Будем реализовывать защиту комплексным методом, на основе некоторых интерфейсов и классов, которые предоставляет платформа .Net. Каждая программа – это поток (или несколько), который имеет информацию о пользователе и о его правах. Достучаться до этой информации можно через свойства потока. Оформим это в виде метода:

```
public class SecurableClass {
        // some code…

        public bool IsInRole(string role) {
            return Thread.CurrentPrincipal.IsInRole(role);
```

```
        }
    }
```

Свойство **CurrentPrincipal** можно устанавливать где угодно в коде, т.е. я хочу сказать надо будет устанавливать значение при входе пользователя в систему (запуске вашей программы). При работе с тестами тоже будет удобное свойсто, т.к. можно выставить любую роль из теста.

CurrentPrincipal имеет тип IPrincipal, в котором объявлен метод IsInRole. Собственно этот метод мы используем и надо переопределить в своем классе. Таким образом необходима своя реализация интерфейса IPrincipal. Сделаем это следующим образом:

```
public class VTPrincipal : IPrincipal {
        private readonly List<string> roles;
        private readonly GenericIdentity identity = new
GenericIdentity("Sys");

        public VTPrincipal(params string[] roles) {
            this.roles = roles.ToList();
        }

        public bool IsInRole(string role) {
            return roles.Contains(role);
        }

        public IIdentity Identity {
            get { return identity; }
        }
    }
```

В целом все готово для написания аспекта и проверки его работы. Проверять на этот раз будем через тесты.

Первый тест будет направлен на то, что администратор системы может записывать и читать значения из защищенных полей.

```
[TestMethod]
        public void AdminCanReadWriteValues() {
            Thread.CurrentPrincipal = new VTPrincipal("Admin");

            var domainClass = new SecurableClass();

            domainClass.SecureField = "new value 1";
            domainClass.SharedField = "new value 2";

            Assert.AreEqual("new value 1", domainClass.SecureField);
            Assert.AreEqual("new value 2", domainClass.SharedField);
        }
```

В данном случае роли указываются строковыми значениями, что не очень хорошо будет при рефакторинге и изменеии кода. Лучше будет использовать enum. Но это можете рассматривать как домашнее задание.

В тесте мы указываем роль, с которой будет запущен поток и создаем класс. Задаем значения свойствам класса и читаем их. Никаких исключений возникнуть не должно, все должно

записаться. Хех, тест в таком виде конечно пройдет, поскольку ничего и не проверяется на запрещение записи. Необходимо закомментировать первую строчку теста с выставлением прав и добиться того, чтобы тест падал. Сейчас мы этим и займемся.

Начинаем создавать класс аспекта. Наследоваться он будет от OnFieldAccess (или **LocationInterceptionAspect** во второй версии) и перегружать будет методы **OnGetValue** и **OnSetValue**.

```
[Serializable]
    public class SecureData : LocationInterceptionAspect {

        public override void OnGetValue(LocationInterceptionArgs args) {
                base.OnGetValue(args);
        }

        public override void OnSetValue(LocationInterceptionArgs args) {
                base.OnSetValue(args);

        }
}
```

Вот такая заготовка. Из передаваемого аргумента можно получить экземпляр класса, значение которое хранится в поле и которое туда будет записано. Нам необходим будет сам экземпляр класса и доступ к его методу **IsInRole**. Для того, чтобы получить этот метод, необходимо будет привести переменную args.Instance к типу нужного класса. Приводить к типу SecurableClass класса будет нехорошо, т.к. этот аспект планируется использовать не один раз. Получается, что надо выделить интерфейс, в который внесем единственный метод **IsInRole**.

```
public interface ISecurable {
        bool IsInRole(string role);
    }

public class SecurableClass : ISecurable { … }
```

Для того, чтобы задать роли доступа, используем конструктор класса. Будем использовать 2 переменные, где зададим права на чтение и запись. Для простоты, роли будут перечислены через запятую, после чего мы распарсим строку в массив.

```
[Serializable]
    public class SecureData : LocationInterceptionAspect {
        private readonly List<string> readRoles;
        private readonly List<string> writeRoles;

        public SecureData(string readRoles, string writeRoles) {
            this.readRoles = readRoles.Split(new[] {","},
StringSplitOptions.RemoveEmptyEntries).ToList();
            this.writeRoles = writeRoles.Split(new[] {","},
StringSplitOptions.RemoveEmptyEntries).ToList();
        }



// other code here

}
```

После этого уже можно писать код в перегруженных методах. Будем брать экземпляр класса, приводить его к нашему интерфейсу и проверять на доступные роли. Т.е:

```
public override void OnGetValue(LocationInterceptionArgs args) {
            var instance = ((ISecurable) args.Instance);
            if (readRoles.Any(instance.IsInRole)) {
                base.OnGetValue(args);
            }
            else {
                throw new SecurityException("You are not authorized to get
value");
            }
        }
```

Для записи значений все проделаем точно так же. После того, как это написали, можно применить аспект к интересующему нас классу.

```
public class SecurableClass : ISecurable {
        public string SharedField { get; set; }

        [SecureData("Admin,User", "Admin")]
        public string SecureField { get; set; }

        public bool IsInRole(string role) {
            var principal = Thread.CurrentPrincipal;
            return principal.IsInRole(role);
        }
    }
```

Запускаем тест и должны увидеть как тот упадет. После чего убираем комментарий с задания роли, перезапускаем тест и он должен проходить. Если все так, то продолжаем. Необходимо написать тест для проверки доступа простому пользователю.

Кроме этого, реализуем другую полезную особенность PostSharp.Можно написать проверку допустимости применения аспектов к полям. Естественно, надо декларативно указать что, аспект применяется только к полям, но этого недостаточно в нашем случае. Т.к. мы ожидаем вполне определенные классы, с конкретными методам, то надо проверить легитимность применения аспекта. Для этого есть у каждого типа аспекта метод **CompileTimeValidate**. С помощью этого метода можно проверить условия применения и в случае чего остановить компиляцию. В нашем случае, надо проверить реализует ли класс интерфейс ISecurable, в противном случае остановим сборку приложения.

```
public override bool CompileTimeValidate(LocationInfo locationInfo) {
            if (!locationInfo.IsStatic &&
                typeof
(ISecurable).IsAssignableFrom(locationInfo.DeclaringType)) {
                return true;
            }

            Message.Write(SeverityType.Error, "AUTHORIZE01",
                          "Cannot apply SecuredDataAttribute to member
{0}.{1} because " +
                          "it is not an instance member of a type
implementing ISecurable",
                          locationInfo.DeclaringType.FullName,
locationInfo.Name);
            return false;
        }
```

Попробуйте теперь удалить наследование от интерфейса и попытаться скомпилировать приложение, ты вы увидете ошибку компиляции. На мой взгляд, это отличная возможность платформы PostSharp. =)

Пробуйте, экспериментируйте и все будет хорошо! =)

Hard’n’heavy!