You can test the logic of your MVC controllers that work with content items, such as articles or products, info objects as well as any other objects. This is done by providing fake representations of these objects. Using this approach, you can create the necessary objects without accessing the database.

The recommended approach for providing fake objects is to use repositories (adapters) that, for content items, make use of generate classes. Repositories provide the following advantages over using document query directly in controllers:

- Your tests will work regardless of the used methods. This is important as not all document query methods are supported when creating fake objects.
- Repositories only need to contain the methods the application really needs. This allows you to make changes to the implementation in a much smaller scale than if you were using document query.

We also recommend writing individual controllers to use dependency injection and retrieve their dependencies via their constructors. This approach allows you to create classes with clearly defined responsibilities, which are, in turn easier to test.

You can also use containers such as Autofac (or similar container implementations) to simplify the process of creating controllers and their dependencies, as well as handling the life cycle of the dependencies.

See also: Initializing Kentico services with dependency injection

See our MVC Demo site for a reference on how to implement tests for your controllers.

Providing content items in controller tests

This is an example showing how the ArticleController controller and ArticleControllerTests are implemented in a similar way as on the sample MVC Demo site. The controller displays a list of articles and their detail. Note that the sample implementation is dependent on external libraries, such as Autofac, TestStack.FluentMVCTesting, and NSubstitute.

1. Create a repository that provides access to content items using their generated providers. The repository only needs to contain the methods required by the application, not the data layer.

```
public interface IArticleRepository
{
   IEnumerable<Article> GetArticles(int count = 0);

   Article GetArticle(Guid nodeGuid);
}
```

```
public sealed class ArticleRepository : IArticleRepository
...
    public Article GetArticle(Guid nodeGuid)
        {
            return ArticleProvider.GetArticle(nodeGuid, mCultureName, mSiteName);
        }
...
```

2. Register the repository in the container configuration. For example, the Kentico MVC Demo site configures the container in the application's Global.asax.cs file.

```
builder.Register<ArticleRepository>().As<IArticleRepository>();
```

3. Use container and any other dependencies in the controller constructor.

```
public class ArticlesController : Controller
{
    private readonly IArticleRepository mArticleRepository;

    public ArticlesController(IArticleRepository repository)
    {
        mArticleRepository = repository;
    }
...
```

4. The controller contains a method that returns a view based on article's NodeGUID or a new instance of the *HttpNotFoundResult* class if the article doesn't exist.

```
// GET: Articles/Show/{guid}
public ActionResult Show(Guid guid)
{
    var article = mArticleRepository.GetArticle(guid);
    if (article == null)
    {
        return HttpNotFound();
    }
    return View(article);
}
```

5. Set up the controller test and create the controller with the required dependencies. The following example uses the NUnit testing framework.

```
[SetUp]
public void SetUp()
{
        // Allow creating of articles without accessing the database.
    Fake().DocumentType<Article>(Article.CLASS_NAME);

        // Mock a repository and set a return value.
        var article = TreeNode.New<Article>().With(a => a.DocumentName = "Test");
    var repository = Substitute.For<IArticleRepository>();
    repository.GetArticle(1).Returns(article);

    mController = new ArticlesController(repository);
}
```

6. Create individual test cases for the controller.

```
[Test]
public void Show_WithoutExistingArticle_ReturnsHttpNotFoundStatusCode()
{
        mController.WithCallTo(c => c.Show(2))
                .ShouldGiveHttpStatus(HttpStatusCode.NotFound);
}
```