

BoardGameSleeves.com

Arcane tinmen

Brian Munksgaard & Jens Christian Rasch | EAAA Backend | December 2016

CONTENT

[INTRODUCTION 1](#_Toc469850173)

[ASP.NET MVC 1](#_Toc469850174)

[Architecture 1](#_Toc469850175)

[Asp.Net MVC Components 2](#_Toc469850176)

[Conventions 4](#_Toc469850177)

[IMPLEMENTATION 6](#_Toc469850178)

[Visual ELEMENTS 6](#_Toc469850179)

[Page layout 6](#_Toc469850180)

[Sketches 6](#_Toc469850181)

[List Sleeves View (public area) 7](#_Toc469850182)

[Cart Information (public area) 7](#_Toc469850183)

[Checkout View (public area) 8](#_Toc469850184)

[Product details view (public area) 8](#_Toc469850185)

[Product list view (admin area) 9](#_Toc469850186)

[Create product view (admin area) 9](#_Toc469850187)

[Grid 10](#_Toc469850188)

[Uml Diagrams 10](#_Toc469850189)

[Domain Model 10](#_Toc469850190)

[Mapping between controllers and views 10](#_Toc469850191)

[Layout and Views 10](#_Toc469850192)

[Model validation 11](#_Toc469850193)

[Entity Framework 11](#_Toc469850194)

[Database migration 11](#_Toc469850195)

[Seeding database data 12](#_Toc469850196)

[REPOSITORIES 12](#_Toc469850197)

[CONCLUSION 12](#_Toc469850198)

# INTRODUCTION

Arcane Tinmen are in the process of updating their online presence and as part of this work, they want to split their current website up, so that they have different websites for each of their brands. To begin this work, they have asked for an updated website for the brand Board Game Sleeves, to which they have acquired a corresponding domain name (boardgamesleeves.com).

This paper will contain two parts where the first will go through the aspects of an ASP.NET MVC website, the components and conventions that ASP.NET MVC is built upon as well as some theory behind building a website using this framework. The second part of the paper will be a detailed description of how we have used these components and conventions in our implementation of a new website for boardgamesleeves.com as well as a walkthrough of the functionality that we have implemented.

Along with this paper, we have included the code for the website (in a Visual Studio solution) as well as the database with some default data (for testing purposes).

# ASP.NET MVC

As mentioned in the introduction we will use the ASP.NET MVC framework to create the Boardgamesleeves.com website. In the following sections, we will describe the components that comprise the framework and some conventions that the framework builds upon.

## Architecture

ASP.NET MVC is built upon an architecture that is created to help structuring the code being developed as well as support the DRY principal. The architecture is using “convention above configuration” which means that it is using conventions for the folder structure and naming of the different components.

An MVC website consists of 3 fundamental components:

* Controllers
* Views
* Models

Models represents the data of the application along with any business logic there might be.

Views are templates for generating dynamic HTML for showing the UI of the application. They define how the look and feel of the website should be and how the different components should be placed in relation to each other. A view can either be loosely or strongly coupled. A strongly coupled view means that there is a tight coupling with the model, which the view can use in its layout and displaying of data.

Controllers are the coupling between the models and the views. Controllers handle the HTTP requests as well as fetching the data from the models and sending the relevant data to the views. Controllers can also contain logic to choose between different views or to map Model data to a ViewModel.

## Asp.Net MVC Components

Apart from the three basic components of an MVC application mentioned above, several other components are a part of the ASP.NET MVC framework. In the table below, we will briefly mention these and explain their usage as well as the reason for them being in the framework.

|  |  |
| --- | --- |
| **Component** | **Description** |
| Controller | Controllers are the binding between Views and Models. The controllers handle the incoming URL requests, fetch the data through the models and send the relevant data through to the Views. The controllers can also contain logic for choosing which View to show or mappings between Models and ViewModels. |
| View | Views are templates for creating dynamic content. They define how the pages look and how the different components should be placed relative to each other. |
| Partial View | A partial view contains small parts that can be included inside a View. It is especially useful for different parts of a website that is used multiple places, like a shopping cart. |
| HTML Helpers methods | HTML Helpers are a way of combining parts of markup and code in such a way that they can be reused throughout the MVC application. There are 2 types of helpers: Inline, Custom and Built-in helpers.  Inline helpers are all created inside the view where they are being used, and are specified by using the @helper tag.  Custom helpers are helper methods that you write yourself and can reuse in multiple Views. These helpers are created either by creating an extension method for the HtmlHelper class or by creating a static method in a utility class.  Built-in helpers are helper methods that are delivered as part of the MVC framework to make creating views easier for the developer. These helpers exist in 2 forms: Standard helpers and Strongly Typed helpers. The strongly typed helpers have a strong coupling between the model and the field in the View. |
| Child Actions | A child action is a controller method that you can call from inside a View and you can set it up so you can only call this type of method from within Views.  Another big difference between child actions a “normal” controller action is that a child action only return a small part of some markup whereas a “normal” action can return an entire View.  Child actions are a good way to create widgets or smaller parts of a markup, that can be reused through a website. |
| Routes | The routing system in ASP.NET MVC specifies how incoming HTTP requests will be handled and how they are directed to the different controllers.  The different routes can be specified in the RouteConfig.cs file and can contain any number of different routes for the website.  We will describe this in more detail when we describe the routes that is set up for BoardGameSleeves.com later in this paper. |
| Models & ViewModels | Models represent the data that the website is working on, along with any business logic there might be.  The difference between Models and ViewModels is that ViewModels are centered on what is to be shown on a View where the Models are centered on the persisted data in the domain. |
| Layouts | It is possible in an ASP.NET MVC to specify layouts as part of a single page, or have pages use a common shared layout. No matter what is used, the layout specifies how the different components are positioned on the individual HTML pages that the controller send back to the browser. |
| Model Binding (IModelBinder)->State | Model binding is the mapping of data from the HTTP Request to parameters used in the action methods of a Controller. Binding can be done to both simple and complex types. It is possible to create your own model binder logic in classes that extends the IModelBinder interface. |
| Areas | Areas are a way to organize a large project or website into smaller and manageable units. An application can contain multiple areas each defining a small portion of the entire application. An example of areas could be the shopping system or the administration system of a website. |

As mentioned earlier the table above is only a short mention of the different components that are part of the ASP.NET MVC framework. In the rest of this paper, we will describe in more detail, how we have used these components and why we have chosen to use the components that we have.

## Conventions

As we mentioned in the section about architecture, the ASP.NET MVC framework is using a couple of naming and folder conventions that makes the building of ASP.NET MVC websites easier from a developer perspective. These conventions are not something that you have to follow as a developer, but by following them, some functionality will work out of the both without the developer needing to think about it or create specific handling of it in the application.

An MVC application has by default a folder structure like the one below. As with all other convetions it is not a structure that you as a developer has to follow.

|  |  |
| --- | --- |
| Folder | Purpose |
| /Controllers | All controller classes that handles HTTP requests are placed here. |
| /Models | All classes that handles data and business logic are placed in this folder. |
| /Views | HTML templates for handling of the UI is placed in this folder. |
| /App\_Data | If the project needs any data files they can be written to and read from this folder. |
| /App\_Start | Any configuration for things like routing, bundling and web api can be placed in this folder. |
| /Scripts | This folder contains all the JavaScript files and scripts which the website uses. |
| /Content | Any files like CSS, image or other content used by the website can be placed in this folder. |

By default if you follow the folder conventions the framework will automatically be able to figure out which Views corresponds to the Actions in the controllers. That means that the developer will not have to set up any system for making sure the correct View is returned, but can rely on the framework to figure this out.

Another convention is the naming of the controllers. A controller can be named by combining the name of the Model that the controller references with the word “Controller”. By using this convention, the framework and routing system can automatically connect a HTTP request for a URL like <http://mysite.com/Product/Index> to the Index action of a Controller called ProductController.

Tightly coupled with the controllers is the naming of the views. The convention specifies that you name the view like the corresponding action method in the controller, and you place it in the Views folder in a subfolder named after the relevant controller. This will make it possible for the framework to locate the correct view automatically, when an action on a controller is being invoked.

# IMPLEMENTATION

In the rest of this paper, we will describe how we have used the conventions and components to implement the new website for BoardGameSleeves.com. We will give examples of how we have solved different aspects concerning the conventions and components, as well as describe the decisions we have made throughout the project.

For clarity, we have included the final product (in a Visual Studio solution) with this paper.

## Visual ELEMENTS

When building a web site one should carefully consider the visual elements on the site. Depending on:

* The audience/users.
* The device (desktop/mobile/etc.) being used to access the site.
* The usage context.
* The actual site content.
* The message you want to send.

We can determine color, typography, visual hierarchy and which gestalt principles to use. Also, a style guide and a site CSS file should be created.

However, since this is beyond the scope of this project, we have decided to use the all the standards in the Bootstrap CSS library, including the Bootstrap grid system. This way we can quickly get a fair UI up and running. You might consider this project a prototype of the system where the visual branding will be done later in the project.

## Page layout

To determine the general page layout, we used UI design sketches. From the sketches we can see that a page has a header, a body and a footer.

* The header contains the site logo, site title and, if applicable, shopping cart info. In the site administration area, the cart info is not displayed. Clicking on the cart will take us to the cart page.
* The footer currently only contains a simple copyright statement. The footer could also be used to hold contact information and references to Arcane Tinmen social media pages.
* The body has two parts, navigation to the left and content to the right. The navigation buttons are based on the different product categories. Currently, the only category is ‘Sleeves’.

## Sketches

During the sketching phase of this project, we came up with the following layouts for the website. Mind you that these are only sketches and are work in progress. The final product could look very different from these. The sketches may be considered an early paper based prototype. They are used to get us started.

### List Sleeves View (public area)

This view is used to list board game sleeves. Using the ‘Game Search’ textbox the view can be filtered to list only the sleeves that matches the search criteria. The list will show basic information and an image for each product. Next to each product there are two buttons, ‘Add to Cart’ and ‘View Details’.

|  |
| --- |
|  |

When adding a product to the cart, the cart is updated, but the page is not changed.

### Cart Information (public area)

This view is used to list the items in the cart. From here we can remove an item from the cart. From this view we can also continue shopping or go to the checkout page.

|  |
| --- |
|  |

### Checkout View (public area)

In the checkout view, the user enters the shipping details. When the ‘Complete’ button is clicked a receipt page is displayed. The checkout view should probably also have a ‘Continue Shopping’ button.

|  |
| --- |
|  |

### Product details view (public area)

This view is used to display detailed product information including a list of games that matches the current product (sleeve). In the view there are two buttons, ‘Add to Cart’ and ‘Back’. A click on either takes the user back to the list view. Of course, when clicking the ‘Add to Cart’, the product is first added to the cart before returning to the list view.

|  |
| --- |
|  |

### Product list view (admin area)

In the product list view in the administration area, an admin user can add, edit, display or delete an item within the different product categories. Please remember that we have currently only one category ‘Sleeves’.

|  |
| --- |
|  |

### Create product view (admin area)

As the view title implies, this view is used to create new products within the current product category.

|  |
| --- |
|  |

### Grid

The sketches lead us to the following grid:

|  |  |
| --- | --- |
| col-md-12 | |
| col-md-2 | col-md-10 |
| col-md-12 | |

In the html code we use the Bootstrap CSS classes col-md-X to define the grid.

## Uml Diagrams

From the site requirements, the sketches and the web stuff from Arcane Tinmen, we have built the following models and diagrams:

### Domain Model

Below you can see the domain model for the new website.

|  |
| --- |
|  |

### Mapping between controllers and views

We have chosen to map the controllers and views like expressed in the image below.

|  |
| --- |
|  |

## Layout and Views

## Model validation

## Entity Framework

In the implementation of the database for our website, we have used the Entity Framework (EF) package to connect the model classes we have created to the tables in the database. EF is an ORM framework provided by Microsoft to automate database related activities for applications. It enables us as developers to focus on creating the models and annotate the properties and classes, and then EF will translate those models and annotations into tables and relations in the database.

By using EF, we have been able to use the code first approach to developing the website and then let EF handle the work of creating the database and the tables needed for our application to work. One of the <More info coming>

### Database migration

To handle changes in the database during the development of the new website we have enabled and used code first migrations. The benefit of using migrations in our project is that we can make sure that any changes to the models is reflected in the database that is being used – and which is included in our delivered Visual Studio project. Another benefit is that we are able to seed the database with test/demo data, which we have used throughout the development process.

To enable migrations in an MVC project, you need to run a couple of commands in the NuGet Package Manager Console and then make small modifications to the migration configuration, to make sure you are seeding the demo data. The commands you need to enable migrations are:

|  |  |
| --- | --- |
| **Command** | **Explanation** |
| enable-migrations | Will set up the migrations for the project and create the necessary files and folders needed for migration to work. |
| add-migration InitialCreate | Will create a migration file, which the next command will use. The “InitialCreate” is for naming the migration so different migrations can be told apart.  A migration file contains all the necessary commands to update a database with latest changes to the models, or to remove these changes form an updated database. |
| update-database | This command will check the database to see what migrations has already been implemented, and then make sure to update the database to the latest migration.  This command will also run the Seed method which will make sure the data in the database is up to date with the latest changes to tables and models. |

### Seeding database data

As mentioned above by using Entity Framework and its migrations possibilities, we have been able to make sure that the database is seeded with initial data. To seed some data as part of migrations means that you can make sure that data, which is needed for your application to work, will be inserted into the tables during the creation of the database.

The data that we have chosen to include, as part of our solution is data that will reflect a website that is already up and running. Normally you will only seed the data that is necessary, but since our application should be able to show a running system, we have chosen to add some extra data such as a couple of users of the system along with a couple of orders.

Another benefit of using migrations and seeding the database is that we during our development phase have had a possibility of “resetting” that data in the database by running the command “update-database” again. If no changes has been made to any model and no new migrations have been created, this command will just run the seeding of the database, and the way we have set up the seed method, all data will be added or updated which will result in any data that have been altered or deleted will be reset.

## REPOSITORIES

To access the data that is stored in the database we have used the principle of separation of concerns. We have done that by using the Repository and Unit of Work design patterns.

Instead of creating a repository for each entity, a rather cumbersome process, we have created a generic repository interface and class. The generic repository can be used for the basic CRUD operations. If more special operations are needed the repository class can be inherited and extended. The generic repository class has the following methods:

* public Repository(BgsContext db, bool saveChanges = true)
* public T GetItem(int id)
* public IList<T> GetItems()
* public int SaveItem(T t)
* public T DeleteItem(int id)

The methods are fairly straightforward except for the SaveItem(…) method which is used both for create and update. The method is shown below:

|  |
| --- |
| public int SaveItem(T t)  {      if (t.EntityId == 0)      {          dbSet.Add(t);          if (saveChanges)          {              db.SaveChanges();          }      }      else      {          db.Entry(t).State = EntityState.Modified;          if (saveChanges)          {              db.SaveChanges();          }      }      return t.EntityId;  } |

The problem is that on an update we need to know the id key of the item being updated. This is solved by letting the generic repository work on an entity super class named BgsEntity like this:

public class Repository<T> : IRepository<T> where T : BgsEntity

The BgsEntity class uses reflection to find the property that will return the correct id:

|  |
| --- |
| /// <summary>  /// Top level model class. All domain classes that are saved and  /// restored from the database should inherit from this class.  /// The generic repository can then be used to perform CRUD like  /// operations on the descendant class.  /// </summary>  public class BgsEntity  {      /// <summary>      /// Returns the current entity id.      /// </summary>      public virtual int EntityId      {          get          {              return GetEntityId(this.GetType());          }      }        /// <summary>      /// Retrieve entity from current class or base class.      /// </summary>      /// <param name="t"></param>      /// <returns></returns>      public virtual int GetEntityId(Type t)      {          try          {              string typeName = t.Name;              string propertyName = typeName + "Id";              PropertyInfo p = t.GetProperty(propertyName);              if (p == null)              {                  return GetEntityId(t.BaseType);              }              else              {                  return (int)p.GetValue(this, null);              }          }          catch (Exception)          {              return 0;          }      }  } |

So, any model class that is a descendant of BgsEntity can use the generic repository. In an UML diagram it looks like this:

# CONCLUSION

Noget med at vi kunne se muligheden for at lave en Detail-visning af produkter også