# Laboratory Work

## Introduction

The public library here in Jönköping has ordered a website from us through which users should be able to browse the books and authors they have in their internal database. The library staff should also be able to login and apply CRUD (Create, Read, Update & Delete) operations on books and authors (each librarian should have her own account).

To accomplish this, they will give us access to their internal database (a relational database) with information about all their books and authors. We do not have access to it yet, but they have sent us a diagram showing us its internal structure (see Figure 1 below).

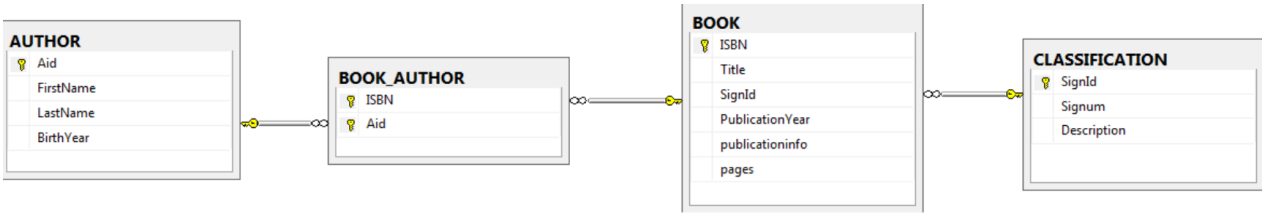


*Figure 1, The design of the library's internal database.*

As usual, we will build this website using a three-layered architecture, as shown in Figure 2 below.

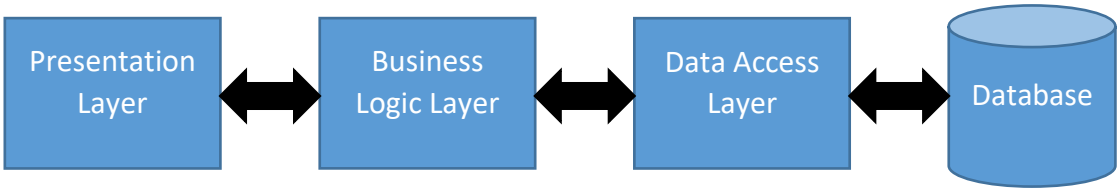


*Figure 2, Overall architecture of our final web application.*

The *Presentation Layer* is responsible for receiving HTTP requests from web browsers and to send back HTTP responses to them. The requests will primarily be handled by telling the *Business Logic Layer* what to do. The Business Logic Layer in turn carries out its job by using the *Data Access Layer*, which in turn writes data to/reads data from the database.

However, the library has well defined milestones they want us to meet, so we will not be able to use this architecture from the beginning, but rather work towards it.

# Lab 1: Creating the graphical user interface

Expected working time: 2 weeks.

The library wants us to start by implementing the graphical user interface for the website. The reason for this is that they want us to quickly deliver a prototype they can test to see if the librarians are happy with the way they will interact with the website. As usual, what the customer thinks of a product is much more important than what we think of the product.

They have sent us a page schematic of the website, showing us how they imagine what the website will look like in the end. This page schematic is shown in Figure 3 below (a bigger picture is provided in a separate file). However, we don't have to follow the page schematic; the page schematic was rather a convenient a way for them to let us know which features they wanted us to implement. Furthermore, the page schematic is not complete, e.g. it doesn't show what the Create new book page should look like, and the edit button should of course only be shown if the user is logged in. And what about pagination? Showing 1.000 books on the same page will not be that convenient.


*Figure 3, The page schematic we received from the library, demonstrating the features we need to implement.*

To implement this prototype, we do not need to worry about the Business Logic Layer, the Data Access Layer nor the database; we only need to care about the Presentation Layer, as shown in Figure 4 below. To implement it, we will use ASP.NET with the MVC pattern (we will find our controllers, views and models in the Presentation Layer (the models will be used in the other layers as well)).

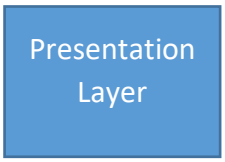


*Figure 4, Our current architecture.*

To make the prototype a bit realistic, we need to populate the graphical user interface with some data. Since we do not have access to a database yet, we can (for now) simply hard code some data (authors, books, etcetera) in lists stored in class variables.

If you're up for it, an even better solution would be to also introduce the Data Access Layer now, and have our data hardcoded in different repositories there, so our current architecture looks like the one in Figure 5 below.

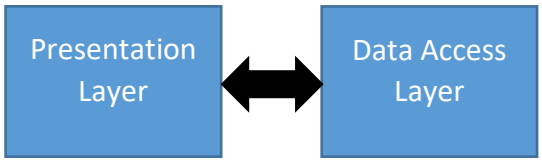


*Figure 5, Our current architecture, if you choose to add the Data Access Layer now.*

This way, we shouldn't have to re-write that much of our code later when we start to make use of a real database. Note that we will only use the MVC pattern in the Presentation Layer.

Try to implement as many features as possible, but it is more important that all different pages exist and that the navigation between them works. For example, implementing a secure member system is not a priority (simply having a form anyone can use to sign in to view the administration pages is good enough), and it is OK if no book is created when submitting a form used for creating books (the important thing is that there exists a form one should be able to use to create a new book).

Here's a list of functionalities I've promised the library an ordinary user should be able to do when we deliver the final application at the end of the project:

- Browse books (show brief information about each book).
- Search for books (show brief information about each book).
- Show detailed information about a specific book.
- Browse authors (show brief information about each author).
- Search for authors (show brief information about each author).
- Show detailed information about a specific author.
- Trying to login as an administrator.

If the user is logged in as an administrator, the user should also be able to:

- Add new books.
- Update information about a specific book.
- Add new authors.
- Update information about a specific author.

- Add new administrators.
- Delete administrators.

The librarians want to try the prototype February 2. By then, it must be good enough for demonstration, so the librarians can navigate it using their own computers. To make this happen, you also need to upload the website and have it up and running on Amazon Web Services. Instructions for uploading a website to AWS is provided in the document `amazon-web-services-guide.pdf`.

## Extra work for grade 4 & 5

On the pages that shows a list of books, authors or administrators, somehow use pagination, e.g. show 20 items on each page, or show all the items that begins with A on the first page, all items that begins with B on the second page, etc.

## Lab 2: Using real data

Expected working time: 2 weeks.

The librarians were not entirely happy with the prototype. They are not sure about what they were unhappy with, but they suspect it can be the lack of real data. Therefore, they have given us an SQL file containing the structure of their internal database along with some real data. Our next task is to use this real data in our prototype.

If you haven't done it yet, you now need to introduce the Data Access Layer, so our architecture will from now on look like the two-layered architecture shown in Figure 6 below.

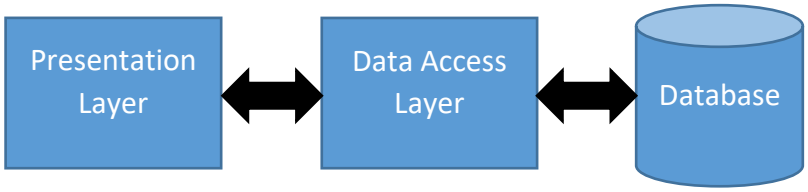


*Figure 6, Our current architecture.*

The Data Access Layer should be a way for the Presentation Layer to read data from and to write data to the database without using any SQL code; the Data Access Layer is the only layer that should contain any SQL code. If we in the future replace the database with another system (e.g. a NoSQL database, or simply store the data as JSON in ordinary files), one should only need to change the implementation of the Data Access Layer; the implementation of the Presentation Layer should still work preciously as it is.

For this project, we will use *Entity Framework* in our Data Access Layer. Entity Framework is an Object-Relational Mapping framework. Through it, we can work with the database without having to write a single SQL query ourselves; we will simply write ordinary C# code, and Entity framework will translate that to SQL queries the database understand.

The librarians want to try this new version of our prototype February 23. By then, we need to have deployed the new version of our website to AWS, as well as have our database up and running at AWS. Instructions for creating a database at AWS and connecting an ASP.NET application to it is provided in a separate document.

### Extra work for grade 4 & 5

The table *Classification* in the database groups books into different categories (books can belong to multiple categories). Add new pages one can use to browse the different classifications and see the books belonging to a specific classification.

## Lab 3: Creating the business logic layer

Expected working time: 2 weeks.

Yea, that was it. Now when the librarians saw familiar authors and books they felt more comfortable, and they have now asked us to complete and deliver the entire website.

Now we have the time to properly build a three-layered architecture by adding the Business Logic Layer, as shown in Figure 7 below.

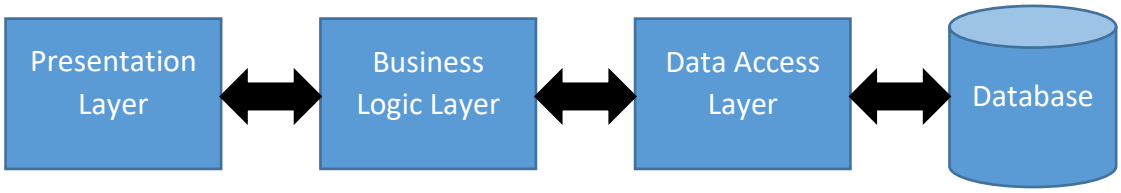


*Figure 7, Our current architecture.*

Some of the code we had in the Presentation Layer (including all code using the Data Access Layer) can now be moved to the Business Logic Layer. The idea with the Business Logic Layer is that if we add another Presentation Layer (a Presentation Layer that's not using HTML & HTTP), we should not need to have the same/similar code in the two different Presentation Layers. This usually includes validating the input. For example, no matter how a user asks the system to add a new book, the system should only add it if it has:

- A proper title (not empty).
- An existing author.
- A positive number of pages.
- Etc.

Furthermore, by separating the graphical user interface from the business logic, it is easier to write tests. Testing functionality through a graphical user interface is usually quite painful and a very time consuming process.

It is up to us to come up with the validation rules. Many of them will be obvious, as the three samples listed above, but some will be less obvious. For example, do we require a *publication year* for a book? Do all books have a known publication year?

The library wants us to deliver an alpha version of the website by March 9. By then all functionality needs to be fully functional, except the security part. Security will be added for the beta release (we don't expect the librarians will try to hack their own website). The new versions should also be deployed to AWS, so the librarians easily can test it using their own computers.

# Lab 4: Adding security

Expected working time: 2 weeks.

The library is now happy with all functionality, so everything that remains is adding security. For this, we need to implement a proper account system, so we can validate users' actions using authentication and authorization (e.g. only authorize librarians that are logged in to create new books).

Usually, we would just add a new table to the database, tell Entity Framework that the database has changed, then ask Entity Framework to update its view of our database, and then work with the database in the same way as we have done with books and authors. An experienced developer would actually do nothing of that on her own, but simply use *Identity* in ASP.NET instead, and let Identity do those parts for her.

However, in order for you to get experience of implementing your own secure member system, you will not do it like that. Instead, you will simply create the new table in the database and use traditional C# classes to communicate with the database, without using Entity Framework. You need to make sure that you are protected against SQL injections, and that the librarians' passwords are stored in a secure way, so no one can read them.

The library wants to have this beta release delivered by March 23. By then, the entire system has to work as it should, because the library is planning to put the system online only a few days later. Again, the new version should be deployed to AWS.

## Extra work for grade 4 & 5

Different administrators should be authorized to do different things. All administrators should be able to administer books and authors. Some of the administrators should also be able to give other administrators a new password (in case they forget it), and some (the most powerful) administrators should be the only ones allowed to create and delete administrators.

## Extra work for grade 5

Some administrators should be able to add new classifications and change existing classifications. This ability should be configurable on individual level. If no books belong to a classification, it should also be possible to delete a specific classification (permanently).

## Extra work for grade 5

Enable HTTPS, so all traffic between the clients and the server is encrypted. Enabling it for local development in Visual Studio is quite easy, just do as shown in Figure 8 below.

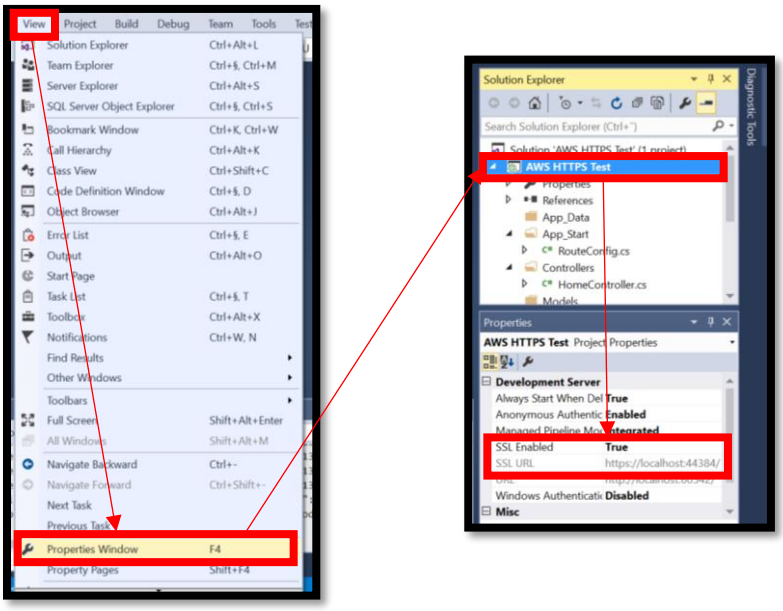


*Figure 8, Enabling HTTPS for an ASP.NET application running on IIS Express from Visual Studio.*

After you have changed SSL Enabled from `False` to `True`, you should be able to use HTTPS the next time you run your web application locally on your computer by using the URL at SSL URL.

**Note:** By default, HTTP uses port 80 and HTTPS uses port 443. However, IIS Express will use other ports for your web application, in case your computer already acts as a server for another web application. When you later deploy your application to a real web server, the default ports will be used.

When you run your web application next time, you will see a confirmation box like the one shown in Figure 9 below.

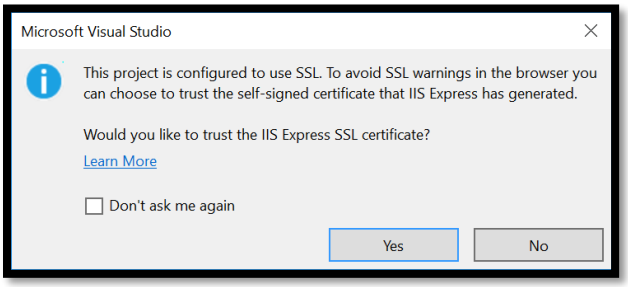


*Figure 9, You're asked to add you web application's certificate to your computer's list of trusted certificate authoroites.*

If you do not want your web browser to warn you that your web application is using an untrusted certificate, choose Yes, otherwise No.

To complete this extra task, you need to create a certificate and use on your web application on AWS. Instructions for doing this can be found in Amazon Web Services Guide. Certificates needs to be signed by a certification authority in order to be trusted, but this usually costs money, so for this project, you are not required to get your certificate signed by a certification authority.

## Lab 5: Building a widget

**This part is only required if you want to get grade 5.**

Expected working time: 1 week.

The library has put the system online, and everything is working great. Hurray! However, not that many people are aware of that the website exists, so not that many people are using it. The librarians think this problem can be solved by putting smart widget on other websites through which users can search for books directly from that other website, and then click on a book title to go to the library's website where they can read more about the book.

The library has asked us to implement this widget for them. We can create the GUI for the widget using HTML and CSS, but to dynamically get data from our server we need to use JavaScript and AJAX. Therefore, we need to create a new "Presentation" Layer acting as an API the widget can use to get the data it needs (in XML or JSON). This new architecture is shown in Figure 10 below. The API is preferably implemented as a REST API.

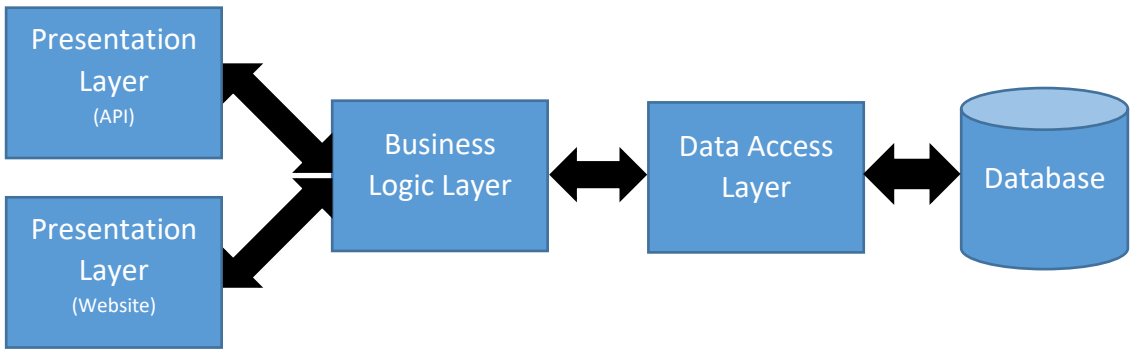


*Figure 10, Current architecture.*

To implement this, we should not need to make any changes to our existing layers; we should only need to add code in the new Presentation Layer. The widget we can simply implement in an ordinary standalone HTML file we send to the library (not part of any of these layers).

To make this work, we also need to enable CORS on the server, otherwise we will not be able to communicate with our server from the widget using the `XMLHttpRequest` object in JavaScript (the same-origin policy forbids us). Actually, an alternative is to use JSONP; use whichever method you find most convenient.

If we choose to accept this extra task, we need to deliver it no later than March 31.