UpStage Team

Investigation of Software Architectures

UpStage 2015 S2

Table of Contents

[Research Criteria 2](#_Toc430606796)

[MVC 3](#_Toc430606797)

[Microservices Architecture 6](#_Toc430606801)

[Monolithic Applications 8](#_Toc430606802)

[Naked Objects 10](#_Toc430606803)

[Event-Driven 12](#_Toc430606804)

[Rationale for Choosing MVC and Event-Driven architecture 13](#_Toc430606805)

# Research Criteria

1. Maintainability  
   - How easily the system can be maintained with minimal risk of breaking something
2. Modularity  
   - How dependent are the system’s components on each other
3. Extendibility  
   - How easy is it to add features into the system
4. Testability  
   - How easy is it to perform tests and are there tools or software which can aid testing
5. Learnability  
   - How easy is it for the project team to learn how to use the architecture properly

# MVC

Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

Model–view–controller has been widely adopted as architecture for World Wide Web applications in major programming languages. Several commercial and noncommercial web application frameworks have been created that enforce the pattern. These frameworks vary in their interpretations, mainly in the way that the MVC responsibilities are divided between the client and server.

### Controller

The **Controller** manages the user requests (received as HTTP GET or POST requests when the user clicks on GUI elements to perform actions). Its main function is to call and coordinate the necessary resources/objects needed to perform the user action. Usually the controller will call the appropriate model for the task and then selects the proper view.

### Model

The **Model** is the data and the rules applying to that data, which represent concepts that the application manages. In any software system, everything is modeled as data that we handle in a certain way. What is a user, a message or a book for an application? Only data that must be handled according to specific rules (date can not be in the future, e-mail must have a specific format, name cannot be more than x characters long, etc).

The model gives the controller a data representation of whatever the user requested (a message, a list of books, a photo album, etc). This data model will be the same no matter how we may want to present it to the user, that's why we can choose any available view to render it.

The model contains the most important part of our application logic, the logic that applies to the problem we are dealing with (a forum, a shop, a bank, etc). The controller contains a more internal-organizational logic for the application itself (more like housekeeping).

### View

The **View** provides different ways to present the data received from the model. They may be templates where that data is filled. There may be several different views and the controller has to decide which one to use.

A web application is usually composed of a set of controllers, models and views. The controller may be structured as a main controller that receives all requests and calls specific controllers that handle actions for each case.

There are a lot of frameworks that provide a basic MVC architecture that you can use. A couple of examples:

* .NET - ASP.NET MVC
* Java - Spring MVC
* Java - JSF
* PHP - Zend Framework
* PHP - CakePHP
* PHP - Symfony
* PHP - CodeIgniter
* Ruby - Ruby on Rails
* Multi-language - PureMVC

**Maintainability**

Pros:

MVC framework has three kinds of components. (Model, View and Controller) Each part is independent to others. Each part can be modified without interact other two components.

**Modularity**

Pros:

The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface.The model directly manages the data, logic and rules of the application. A *view* can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the controller, accepts input and converts it to commands for the model or view.

**Extendibility**

Pros:

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

**Testability**

Pros:

The code is very easy to test. Most of frameworks which I list at the begging have inherent testing for implementation.

# Microservices Architecture

**Maintainability**

Pros:

* Easier to scale development. It enables you to organise the development effort around multiple team members. If each member is responsible a single service, the team can develop, deploy and scale their service independently of all of the other members.
* Allows you to release smaller change sets. A one line change to a hundred thousand line monolith application requires the entire application to be deployed. A one line change to a microservice only requires the service to be deployed.

**Modularity**

Pros:

* Each microservice is relatively small
* Each service can be deployed independently of other service.
* Improved fault isolation. Other services will not be affected and will continue to handle requests.
* Separation of concerns and isolated functionality within the codebase

**Extendibility**

Pros:

* Easier to deploy new versions of services frequently.

**Testability**

Cons:

* Developer tools or IDEs are oriented on building monolithic applications and don’t provide explicit support for developing distributes applications. Therefore testing is more difficult.
* Refactorying difficulty as interfaces and application boundaries are spread across the microservices.

**Learnability**

Pros:

* Easier for a developer to understand as each microservice is relatively small.
* Easier as it allows you to focus on scaling just those services that need scaling and not the whole application.

Cons:

* Deployment complexity. In production, there is the operational complexity of deploying and managing a system comprised of many different service types.
* Implementing use cases that span multiple services requires careful coordination between the teams.

**Extra**

Pros:

* The IDE is faster making developers more productive.
* The web container starts faster.

Cons:

* Increased memory consumption.

**References**

<http://microservices.io/patterns/microservices.html>

<https://www.madetech.com/blog/microservices-pros-and-cons>

# [Monolithic Applications](https://www.facebook.com/notes/upstage/software-architecture-monolithic-applications/1493642404263734)

(Single tier application eg. MVC) A software system is called "monolithic" if it has a **monolithic architecture**, in which functionally distinguishable aspects (for example data input and output, data processing, error handling, and the user interface), are not architecturally separate components but are all interwoven. (<https://en.wikipedia.org/wiki/Monolithic_system>)

**Maintainability**

+ Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications (faster initial development)

- Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are.

- Large code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It's a downwards spiral. ([http://microservices.io/patterns/monolithic.html](http://l.facebook.com/l.php?u=http%3A%2F%2Fmicroservices.io%2Fpatterns%2Fmonolithic.html&h=JAQHHceFI&enc=AZOKmxLtNWiab4P9RsUfE__EWxrw2OJJDMHe22-yznr4Pa_Y10qJT0DJQHpDjhscQEXE_V6nD9NKjLnLv7iM_otL6alE029ILnvi7-5oYyGWzGF0RbFZjjtynCgzpk0KmIZDH7ysB6nyPc-zsdm-mTFlKgWtZc0AzVX5_aT0vgpKDQ&s=1))

**Modularity**- This approach is used for applications that do not require modularity.

-  There are no hard module boundaries so modularity breaks down over time due to large code base. (whereas microservices is modular and will have independent services that are developed, tested and deployed separately so it can be extended much easier than monolithic)

**Extendibility**

-Not extendable because its not made for modularity.  (opposite of microservices)

+ Supports a variety of different clients including desktop browsers, mobile browsers and native mobile applications (<http://microservices.io/patterns/monolithic.html>)

-Monolithic servers, once initial architecture is made, do not leave much space for innovation. ([http://technologyconversations.com/2015/01/07/monolithic-servers-vs-microservices/](http://l.facebook.com/l.php?u=http%3A%2F%2Ftechnologyconversations.com%2F2015%2F01%2F07%2Fmonolithic-servers-vs-microservices%2F&h=vAQGDgc8R&enc=AZOn_GJ4hdhYFAMgLrxdlCPSwY91e_Stxq0jzqL08CWXyPoddeAIshm1SWsnrlC5HXhY7ov-89nPw5CQEUindSrC60eXxnRi1aoGLCili7Z96IwPifH8I3LW2U9s_g8mDsrZsitc25ihxJIGzv77SMFG74UInzYmE8fWGBWDMqbIVg&s=1))

-The larger the coding gets the harder to implement change and have iterations

**Testability**

- changing things takes time and experimentation is very risky since it potentially affects everything.

**Learnability** - Is it easy to understand the structure of the software architecture?

- High set-up costs :- the larger the application, the more difficult it would be for new developers to be productive. They would have to understand the infrastructure of whatever they do, so this could be a pretty big issue.

**Performance**

-Overloaded web container - the larger the application the longer it takes to start up. It also slows down deployment too.

Changing things takes time and experimentation is very risky since it potentially affects everything.

# Naked Objects

The naked objects pattern is defined by three principles:

1. All business logic should be encapsulated onto the domain objects. This principle is not unique to naked objects: it is just a strong commitment to encapsulation.
2. The user interface should be a direct representation of the domain objects, with all user actions explicitly consist in the creating or the retrieving of domain objects and/or invoking [methods](https://en.wikipedia.org/wiki/Method_%28computer_science%29) on those objects. This principle is also not unique to naked objects: it is just a specific interpretation of an object-oriented user interface (OOUI).  
   The original idea in the naked objects pattern arises from the combination of these two which form the third principle:
3. The user interface shall be 100% automatically created from the definition of the domain objects. This may be done using several different technologies, including source code generation; implementations of the naked objects pattern to date have favoured the technology of reflection.

**Maintainability**

Cons:

All the objects which created by the Naked Objects are single layer only. If one part of the object has some bugs, it might lead another error to the other parts of the object. For example the user interface is created automatically from domain objects. If something wrong within the domain objects, the user interface might create in errors. This makes developer hard to detect where the real problem comes from.

**Modularity**

Cons:

The components of the Naked Objects are all connected so it is not modular.

**Extendibility**

Cons:

The Naked Objects are a bit difficult to extend new feature within the current objects, because each components of the Naked Objects are all connected. If we want to extend one part within the Naked Objects, we might need to think about all connection which relate to the extend part.

**Testability**

Cons:

The Naked Objects all components of the object are connected which means it is a bit complicated to testing single component.

**Learnability**

Pros:

The biggest advantage for the Naked Objects are faster developing and easier analyses. It only need to decide what objects are which is easy.

# Event-Driven Architecture

Event is defined as a significant change of state, this can mean things like in the instance of a car going from "For Sale" to "Sold".

Events don't travel, they just occur and emit a message.

The event producer doesn't know that the receiver has received the message/event

Action listeners, and actions preformed

Broadcast communication where events are published as they occur in an asynchronous fashion with single events as opposed to single aggregated events receiving system can have interest in certain types of events.

There is no call stack involved so no waiting for other methods to occur.

No inherent coordination, continuation or context preservation. The call stack prefers a single line of execution and thrives where the caller and callie share the same memory.

More assumptions means more coupling.

The benefits of EDA beyond loose coupling

It is easy to track all of the events that occur in the system and then the state of the system can easily be replicated based on events that have occurred. These events used to recreate a state can be modified to change details of the state.

Could be associated with component based Architecture.

# Rationale for Choosing MVC and Event-Driven architecture

Using the right architecture can greatly increase the success rate of a software development project. Not only will it make the codebase more structured and tidy, it can also reduce potential problems such as coupling. However, not all architecture patterns are suitable for UpStage, and so the project team decided to investigate several options to try and evaluate which architecture would work best for the project.

Each team member researched and investigated an architecture pattern and evaluated it according to the criteria we have set. The criteria are:

The architectures we researched are Model-View-Controller (MVC from here on), Microservices, Monolithic, Event-Driven and Naked Objects. They each have their own advantages and disadvantages, and we found that making use of more than one architecture pattern will be more beneficial to the project, as they can amplify each other’s strengths while covering their weaknesses. The two architecture patterns we think is most suitable for UpStage are MVC and Event-Driven.

MVC excels in all of the criteria we have set. It divides the system into 3 main components; the Model, where the data is stored and processed, the View, which handles the output of the data that is seen by the user, and the Controller, which handles user input and tells the Model and View what to do or change. This ensures that separation of concern is in place and makes it easier to take a modular approach.

In terms of maintainability, architectures such as Monolithic and Naked Objects are relatively harder to maintain as they do not separate the different parts of the code. This can make it hard to maintain the codebase will grow larger and larger and get overwhelming. Microservices and Event-Driven separates the services/features, which makes them maintainable without touching other classes.

In the same vein, the two architecture patterns are also more testable. If something goes wrong or a bug pops up, the source will be in the feature being tested, making it easier to locate. In Monolithic and Naked Objects, it will be hard to locate as something that is broken in another feature can affect the feature being tested.

In terms of how easy it is to implement new features, or extend existing features, a Monolithic or Naked Objects approach can be easier, by virtue of being able to easily modify features that rely on other features. However, Microservices and Event-Driven are safer as they ensure that an existing feature is not broken by the addition of a new feature. The disadvantage is that features that use parts of another feature have to be handled carefully, and both halves of the feature have to be updated.

Both Microservices and Event-Driven can be integrated with MVC for UpStage, but the project team feels that Event-Driven is easier to learn and use. Microservices will require the team to first decide which parts of UpStage is a service, and then create a middle layer which can call the appropriate service when needed. For Event-Driven, there is no middle layer, which makes it less abstracted and easier to use.