UpStage Team

Investigation of Software Architectures

UpStage 2015 S2

Table of Contents

[Introduction 2](#_Toc431208068)

[Research Criteria 4](#_Toc431208069)

[MVC 5](#_Toc431208070)

[Microservices Architecture 9](#_Toc431208074)

[Monolithic Applications 12](#_Toc431208075)

[Naked Objects 14](#_Toc431208076)

[Event-Driven Architecture 16](#_Toc431208077)

[Rationale for Choosing MVC and Event-Driven architecture 20](#_Toc431208078)

[References 22](#_Toc431208079)

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Changes |
| 1 | September 21, 2015 | Alyssa Byun, Jing Han, Joshua Kartono, Siatua Uili, and William Stokes |  |
| 2 | September 30, 2015 | Alyssa Byun, Jing Han, Joshua Kartono, and Siatua Uili | * Introduction added * References added * Diagrams for architecture added * Final architecture diagram added |

# Introduction

Selecting an architecture pattern should solve the structural problems commonly found in software design and for UpStage it currently needs an architecture that supports the real-time cyberformance website in maintaining modularity, security, supporting all functionality and other criteria we have selected for research.

What the chosen architecture will need to support

For better performance and development, UpStage needs to be

* Extendible with regards to its many domain objects (i.e. users, avatars, props etc. mentioned later)
* Security is needed for users’ activities and information. Users should not be able to directly access the data storage.
* Modularity is necessary for loose coupling and for future developers to easily extend UpStage features/functionality and last but not least the architecture will have to support
* Testability for checking that UpStage is working well.

Maintainability, Extendibility & Modularity benefits new students joining each semester and it needs to be easy for them to extend and maintain as this project is ongoing. (Bad example: the current upstage)

The ideal architecture needs a layer for browser, client-side, server side and storage.

The candidates

Martin Eisenbarth, the first developer for UpStage, has been developing since 2011 an architecture called DownStage based on the experiences with UpStage in an attempt to gain flexibility, extensibility and sustainability. His shared (https://trac.foobarlab.net/downstage) work was last updated 3 years ago and due to a lack of resources found on this we have decided not to look further into this architecture. (http://www.foobarlab.net/projects)

The group will be investigating:

* MVC because it is an architecture most commonly used in IDEs and is taught in AUT so everyone is familiar with using this architecture.
* Event Driven architecture is another candidate because of its reputation as the go to pattern for real-time applications which is what UpStage needs.
* Microservices could be useful in UpStage for its componentization of independent services that give the benefits of modularity, extendibility and decentralization that we may find useful to us.

Other candidates that we have minor research on are:

* Monolithic applications and
* Naked objects

Because there are mainly disadvantages found in these architectures they will not be used so we see no need to further investigate monoliths and naked objects as we have with the first three candidates listed.

UpStage main domain objects (bottom up)

The main domain objects identified for UpStage are:

* Users
* Avatars
* Stage
* Props
* Backdrop
* Chat
* Audio
* Text-to-speech
* Video avatar
* Storage

Why not other architectures?

It is important to know why other options have not been investigated for future developers to understand the research done. Other candidates, such as Ruby on Rails, are not researched based on UpStage they may not be suitable or due to the group’s preferences and poor knowledge in other architectures.

# 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.

**MVC in the Upstage**

The domain object of the Upstage consists of Avatar, Audio, and Backdrop etc.  
MVC could easily handle these parts. For example, user could change a new avatar by clicking the image. The controller catch the command and tell model to change the data for the user, after that the observer of model catch the changing and make view display the new avatar to the user based on the particular browser (Google Chrom, Fire Fox etc.)

### 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 cannot 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) There is an observer pattern between Model and View. The observer pattern catches the changing from Model and informs View. Because each component is relatively independent to others, this could make developer easier to maintain and lower the costs in life-cycle. For example, if there is a message could not display in Firefox browser but fine in other browsers, the developer could easily tell the issue happened in view component.

**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.

**Security**

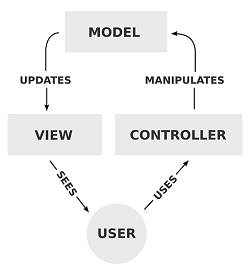
Pros:

The user could not directly query through the database. Every command is caught by controller. Controller could block and stop the command before it send to the model.

**For the Upstage**

Pros:

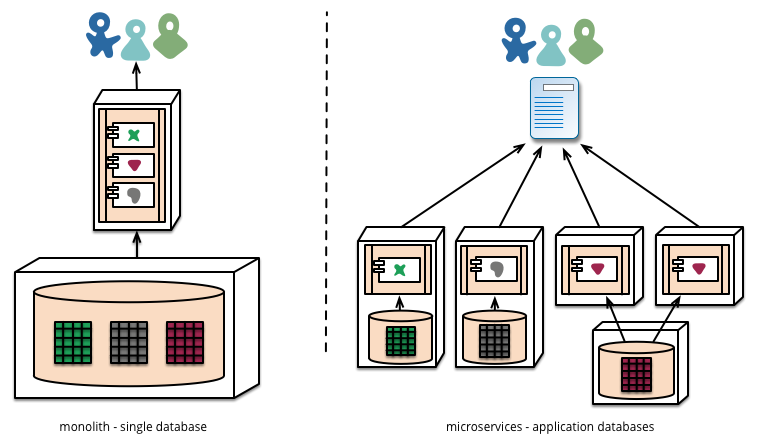
The Upstage is a web-based platform. The MVC architecture pattern is widely used for World Wide Web applications. As the Upstage is an ongoing project, it is better to use software architecture like MVC, this could make other new developers who will work on the Upstage easier to catch up. By using MVC in the Upstage, developer could work on different components separately. Some developers could just work on user interface, some developers could focus on how to processing user requirement or how to dealing with data through model.



# Microservices Architecture

Microservices architecture describes a particular way of designing software applications as suites of independently small deployable services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. There is a bare minimum of centralised management of these services, which may be written in different programming languages and use different data storage technologies.

As UpStage has many functions that users may take actions, each function can be made as a microservice that is called by the centralised class. Each microservice is independent to each other, this prevents the code coupling. Each microservice is small so it provides only one trivial function, which can make this software architecture much more modular. Also, only relative services are called by the centralised class when the system gets requested by user, it uses the minimum resources while some other architectures such as publish-subscribe pattern send messages to every subscribers, which uses more resources than it actually needed even though the resources are lightweight.



This diagram shows the difference between microservices and monolithic architectures. If we apply the microservices architecture for UpStage project, we would probably start to make microservices with a main functions such as a chat service, an audio service, and a visual service for starters to keep things simple. As the whole point of an iterative approach is to improve over time, we could then add other services such as an authentication service, etc. and keep extending the system.

**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.
* Refactoring 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 web container starts faster.

Cons:

* Increased memory consumption. The microservices architecture replaces N monolithic application instances. If each service runs in its own VM, the overhead is higher.

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

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. ([“Monolithic](https://en.wikipedia.org/wiki/Monolithic_system) system”, n.d.)

**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. ([“Microservice](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) Architecture”, n.d.)

**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 it’s not made for modularity.  (Opposite of microservices)

+ Supports a variety of different clients including desktop browsers, mobile browsers and native mobile applications (Richardson, 2014)

-Monolithic servers, once initial architecture is made, do not leave much space for innovation. (Farcic, 2015)

-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

EDA utilizes Events to handle changes in the system. An Event is defined as a significant state of change; for example, an instance of a Car object going from “For Sale” to “Sold” is one such event. Events do not travel; they just occur and broadcast a message to all listeners. In these cases, the event producer does not know if the event receiver(s) have received the event or not. Event receiver(s) will then check if the message is one they are interested in, and if it is, will catch it and handle it. If it is not, the message is simply thrown away. All events are recorded as they happen and these can be used for testing or maintenance purposes. This architecture uses the Observer Pattern, just like MVC; the components are relatively separate, making the whole architecture more modular.

For UpStage, EDA is suitable as messages broadcast from events are not polled; multiple events can broadcast their message simultaneously in an asynchronous fashion. This is important as UpStage needs to be real-time, and having delays when multiple users perform actions is not acceptable. The use of an Observer Pattern also helps as having new team members every semester means the code have as few coupling as possible.

**Maintainability**

Pros:

As event receivers only handles events they are interested in, events that are designated to other receivers will not affect them. As such, new members do not have to worry about previous sender-receiver interactions created, and can focus on new ones. Also, if a bug is found, only the new features added since last maintenance have to be checked, further reducing the range of possible bug sources.

**Modularity**

Pros:

As EDA utilizes an Observer Pattern, where the event sender do not care whether or not the event receiver(s) received their message, the architecture is highly modular. This ties into the maintainability and extendibility criteria as being modular means bug sources are easy to locate.

**Extendibility**

Pros:

Adding a feature to an existing EDA is simple as event generators do not interact with each other. When a new feature is implemented, existing event receivers (or new ones) will simply need to capture the message broadcasted by the new feature’s events. As other event receivers will simply throw this message away, it will only affect event receivers which explicitly accept the message, reducing the scope of errors that can come up. This ensures that newer team members will not break any existing features as they add new ones.

Cons:

When modifying an existing feature, both sender and receiver have to be modified. This can cause strange behaviour if not done properly.

**Testability**

Pros:

EDA has an inherent advantage when it comes to testing; as any changes made in the system are generated from an event, those events can be used to replicate a scenario. Specific events can also be produced to test specific scenarios should the need to do so arises.

The ability to reproduce a specific system state from events is important for new team members as this will allow them to see and change tests that were done before they joined the team. Another advantage is that when a bug occurs during testing, the events that happened prior to the bug can be used to reproduce the bug, making it easier to figure out the source of the bug. This can be documented in the bug reports to ensure that future team members can reproduce a bug(s).

**Learnability**

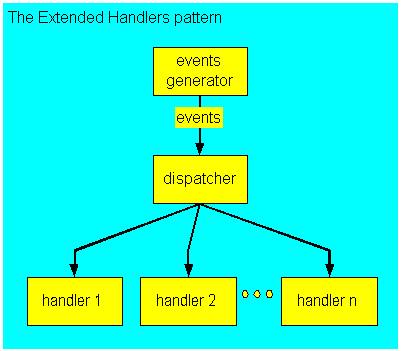
Pros:

EDA is a relatively simple architecture to learn as it is in essence a collection of event sender and listener interactions, which AUT students are taught about in the programming papers. It is also similar to MVC, and can be integrated into a MVC system, which is also taught at AUT. This makes EDA easy to pick up for existing and new members.

**Security**

Pros:

As events automatically broadcast their message, users cannot send their own messages to influence what the system does. Regardless, if a user managed to inject a message, those messages will be thrown away by the event receivers as they only capture and handle events which are explicitly for them.



Each event generator will send its events to the dispatcher, which will then send it to all applicable event receivers. Each receiver then checks if the event was meant for them, and if it is not, throws it away. Otherwise, it handles the event (each handler may handle it differently).

# 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 code 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.

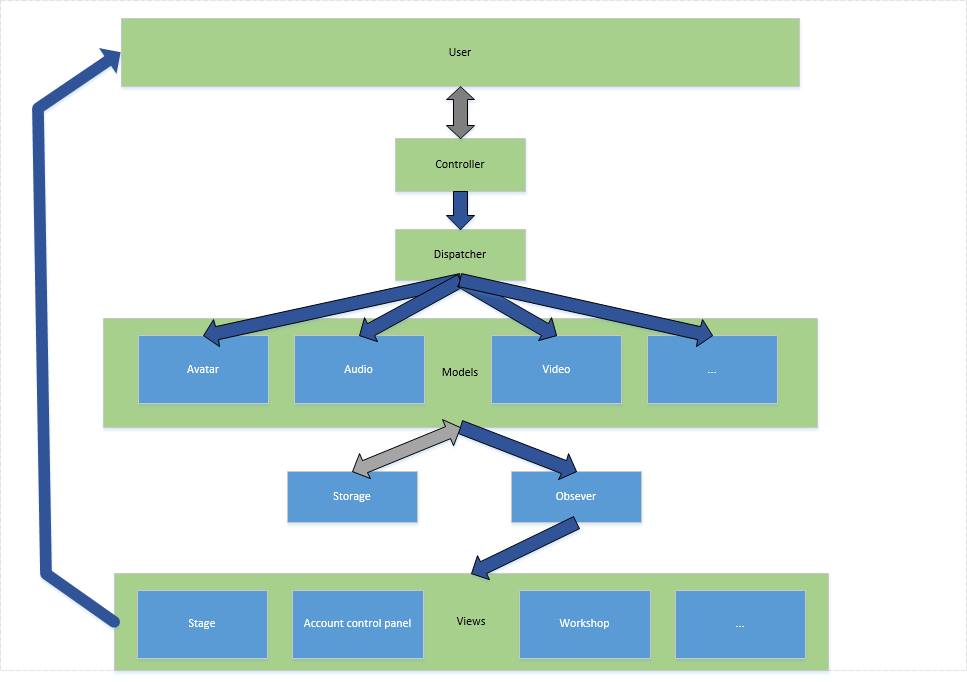
MVC excels in all of the criteria we have set. It divides the system into 3 main components; the Model, where the data is stored and processed, the View, which handles the output of the data that is seen by the user, and the Controller, which handles user input and tells the Model and View what to do 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 code 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.



# References

*Event-driven architecture.* (n.d.) In *Wikipedia.* Retrieved from: <https://en.wikipedia.org/wiki/Event-driven_architecture>

Farcic, V. (2015, Jan 7). *Monolithic servers vs microservices*. Retrieved from <http://technologyconversations.com/2015/01/07/monolithic-servers-vs-microservices/>

MacDonald, R. (2015). *Microservices: Pros & cons of using microservies on a project*. Retrieved from <https://www.madetech.com/blog/microservices-pros-and-cons>

*Model-view-controller*. (n.d.) Retrieved on 28/09/2015, from <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

*Monolithic architecture.* (n.d.). In *Wikipedia.* Retrieved from <https://en.wikipedia.org/wiki/Monolithic_system>

Nadle, B. (2012). *A Better Understanding of MVC (Model-View-Controller) Thanks To Steven Neiland.* Retrieved on 28/09/2015, from <http://www.bennadel.com/blog/2379-a-better-understanding-of-mvc-model-view-controller-thanks-to-steven-neiland.htm>

Richardson, C. (2014). *Pattern: Microservies architecture*. Retrieved from <http://microservices.io/patterns/microservices.html>

Richardson, C. (2014). *Pattern: Monolithic Architecture.* Retrieved from <http://microservices.io/patterns/monolithic.html>

Rouse, M. (2011.) *What is event-driven architecture (EDA)?* Retrieved from http[://searchsoa.techtarget.com/def.../event-driven-architecture](http://searchsoa.techtarget.com/definition/event-driven-architecture)