UpStage Team

Investigation of Software Architectures

UpStage 2015 S2

Table of Contents

[Introduction 2](#_Toc432167646)

[Software Architecture Requirements 5](#_Toc432167654)

[Model–View–Controller 6](#_Toc432167655)

[Microservices Architecture 10](#_Toc432167656)

[Event-Driven Architecture 13](#_Toc432167657)

[Rationale for Choosing MVC and Event-Driven architecture 17](#_Toc432167658)

[Appendix 19](#_Toc432167659)

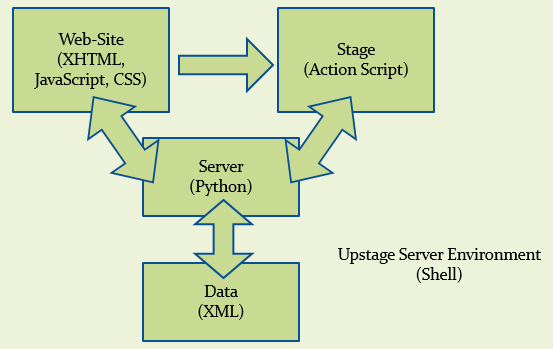
[References 21](#_Toc432167661)

|  |  |  |  |
| --- | --- | --- | --- |
| 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 * Issues addressed from v1 feedback fixed |
| 3 | October 7, 2015 | Alyssa Byun, Jing Han, Joshua Kartono, and Siatua Uili | * Monolithic architecture removed * Naked object removed * Final architecture diagram edited * Issues addressed from v2 feedback fixed |

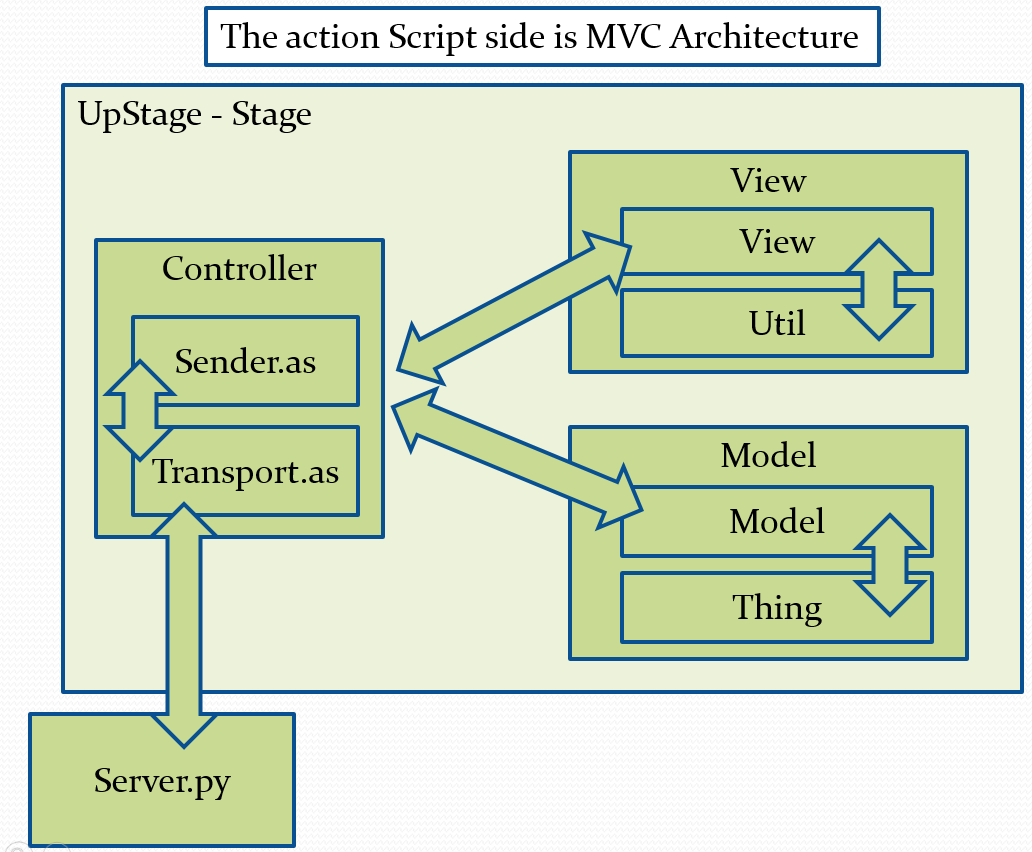
# Introduction

Selecting an architecture pattern should solve the structural problems commonly found in software design and support the functionality required of the software. UpStage 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. At the moment UpStage uses MVC as seen in the Stage architecture diagram below.

Existing System Architecture (from the 2013 UpStage induction powerpoint)



Existing Stage Architecture (from 2013 UpStage induction powerpoint)



UpStage main domain objects (bottom up)

The main domain objects identified for UpStage are:

* User (roles: player, audience, maker, and admin)
* Avatar (image or video)
* Stage
* Prop
* Backdrop
* Media (audio, image, and video)
* Performance
* Chat log

## New architecture requirements

For better performance and development, UpStage needs to be

* Extendibility: adding functions
* Security is needed for users’ activities and information to be protected.
* Modularity is necessary for loose coupling and for future developers to easily extend UpStage features/functionality
* Testability for checking that UpStage is working well.

Maintainability, Extendibility & Modularity benefits new students joining each semester as it needs to be easy for them to extend and maintain the project since it is ongoing.

## Existing UpStage issues

UpStage shows these problems as mentioned in the UpStage 2014 Motivational document:

* It is not modular
* Changed over time by many developers
* Tight coupling
* Hard to refactor
* Coding standards were not followed (e.g. non-strict naming conventions. There is a class called “Thing”)
* Obsolete technology (i.e. ActionScript, Twisted, Flash, etc.)

## The candidates

Martin Eisenbarth, 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 work (https://trac.foobarlab.net/downstage) was last updated 3 years ago and apart from that we were able to get a detailed explanation of his architecture in an email which is included in the appendix. Martin’s email was a response to a request for more information on his architecture and this includes a lot of other concepts which we are not familiar with. DownStage is still in progress and due to a lack of resources found on this we have decided not to look further into this architecture or we would need more time to study his methods.

The group will be investigating:

* MVC because it is an interactive system which is useful in UpStage.
* Event Driven architecture is modular and extendable since it is able to use multiple event handlers for many states. This may be useful for UpStage since its response is near real time (Rouse, 2011).
* 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.

## 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 Monolithic Applications and Naked Objects, have been omitted as they do not meet some of the architecture requirements for UpStage.

There has been some research on Monoliths to compare and contrast with Microservices, it’s opposite, but the research done was irrelevant to UpStage.

Monolithic applications have a structure where its components are all interwoven so does not separate them like Microservices and thus lacks modularity which is why it is not included (“Monolithic system,” n.d.).

As for Naked Objects it also does not allow separation of different parts of the code. This restricts the program from being modular and it will not be easily maintained hence why it will not be studied.

# Software Architecture Requirements

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

# Model–View–Controller

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 the 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 UpStage**

The domain objects of UpStage consist of Avatar, Audio, and Backdrop etc. MVC can easily handle these parts. For example, users can change their avatar to a new avatar by clicking the image. The controller catches the command and tells the model to change the data for the user, after that the registered view catches the changes from model and the view display the new avatar to the user based.

**Controller**

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.

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.

**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? For UpStage, avatars can be modeled. Only data that must be handled using 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.

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

**Observer Pattern**

Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

This allows the number and type of view objects to be configured dynamically, instead of being statically specified at compile-time.

**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 changes from Model and informs View. Because each component is relatively independent to others, this will make it easier for developers to maintain and lower the costs in life-cycle.

**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 inputs and converts them 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. The composite pattern is a partitioning design pattern. The composite object combines its child objects’ functionality. The composite objects and individual objects should be treated uniformly.

**Testability**

Pros:

The code is very easy to test. Most of the frameworks which are listed at the beginning have inherent testing for implementation. It is easier for developers to test a single component in MVC, because the components in MVC are decoupled. The errors will be less likely to come from other components.

**Security**

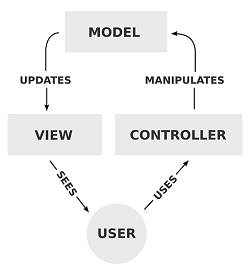
Pros:

The user cannot directly query through the database. Every command is caught by controller. The Controller can block and stop the command before it sends it to the model.

**For the UpStage**

Pros:

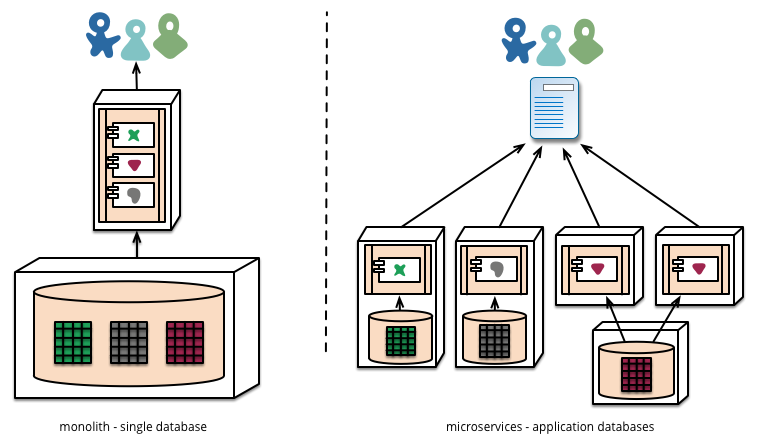
UpStage is a web-based platform. The MVC architecture pattern is widely used for World Wide Web applications. As UpStage is an ongoing project, it is better to use a software architecture like MVC, as this will make it easier for new developers to understand.



# 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 related 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.

# Event-Driven Architecture

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) needs to handle 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; as the event sender(s) and handler(s) do not need to wait for a reply from each other, the system becomes less coupled.

For UpStage, EDA is suitable as messages broadcast from events are not processed serially; 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, more focus can be placed on the features added since the last maintenance, rather than the old features.

**Modularity**

Pros:

As EDA utilizes an Observer Pattern, where the event sender does not care whether or not the event receiver(s) handles 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 behavior 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**

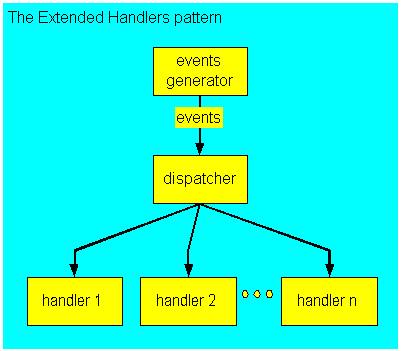
Pros:

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.

We have researched several software architectures including Monolithic and Naked Objects, but each of these architectures could not be the one of the candidates for UpStage architecture investigation. This is because that the 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. Also, it will be hard to locate as something that is broken in another feature can affect the feature being tested.

The architectures we focused are Model-View-Controller (MVC from here on), Microservices, and Event-Driven. 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.

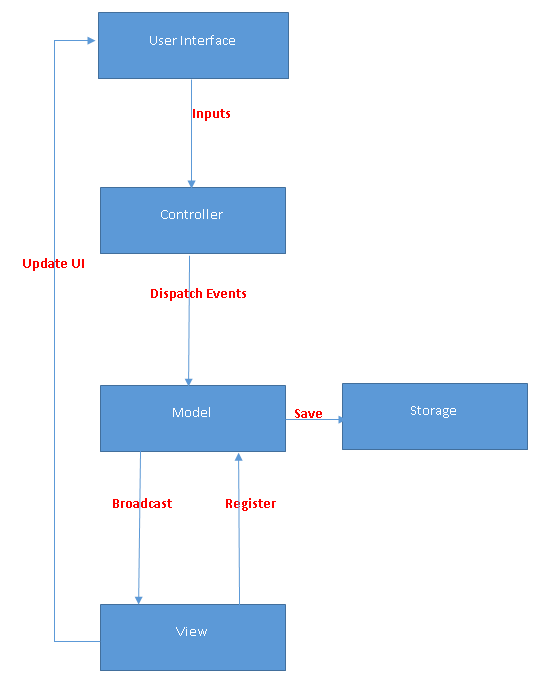
MVC excels in all of the criteria we have set. It divides the system into 3 main components; the Model, where the data (i.e. media files, user information, etc.) is stored and processed, the View, which handles the output of the data that is seen by the user, and the Controller, which handles user input 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, 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.

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.



# Appendix

DownStage Architecture *(Email from Martin Eisenbarth to UpStage)*

Hi Sia,

The \*Trac page unfortunately is outdated. I just have limited time so my plan is to create a new website (including blog, gitlab, bugtracker and wiki) containing more information when the work is more progressed. This will still take some time as I am quite busy doing my regular work. Also I am doing a new server setup for my developments which needs to be done before I continue working on DownStage.

Nonetheless I can unveil in an outline how the architecture looks like:

Basically the core is mainly about message passing. Everything is a message. For this I use right now a functional approach utilizing the Erlang programming language (http://www.erlang.org/). I have made tests before using Maven and Java/Scala with Akka (http://akka.io/). Initially I made experiments with the Axonframework (http://www.axonframework.org/). Akka and Axon both are also message based and can be used for a CQRS architecture (http://martinfowler.com/bliki/CQRS.html). For my approach with Erlang I am also implementing a CQRS pattern. Java respectively Scala has some disadvantages, as it introduces some complexity and requires a high skill level in programming. Those are also quite resource hungry. Erlang is much more simple and lightweight. Instead of having highly complex error handlers for fault tolerance like in Java/Scala (e.g. see the Nextflix Hystrix Library for a short circuit pattern: https://github.com/Netflix/Hystrix), Erlang has the principle of "let it crash" which is much more simple and convenient.

Overall "Separation of Concerns" is essential. The core is therefore just responsible for synchronically and asynchronically passing messages around. CQRS can be combined with "Event Sourcing" which allows to record and playback Events. This also simplifies delayed event handling for different clients.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*Trac – refers to Martins shared work on foobarlab (https://trac.foobarlab.net/downstage)

For databases I had a look at Key-Value Stores and Graph Databases. Both are interessting concepts which allow easy and fast data access. In the end I will have a plain file store, a key-value store, a document store and an optional graph database. Some parts of the storage act like a tuple space (kind of blackboard pattern) and are handled by "Actors" (Erlang processes).

The overall architecture is layered, but more like an onion than a classical three-tier-approach. In the outer region it uses a RPC mechanism and offers api's for different programming language. My approach is therefore mostly language independent and it is planned to allow scripting at least in LUA, Python, Erlang/Elixir and JavaScript.

Other languages should be addable without a big hassle, as the interface to the core is using a RPC model.

Erlang has also some nice benefits like hot code deployment. And it works on low cpu powered devices as high end servers.

In the end the architecture is not verified yet, but during my research I did for quite a while now (since 2009) I am quite sure it is very feasable.

I have also some ideas for the architecture which involve onion-networking and plug-and-play functionality. In the end the server should be able to run on a small device like a Raspberry Pi and discover other server nodes automatically which can then be thought of as a distributed system for handling events. This is not the primary approach but I have this in mind for the extension of the architecture.

Downstage as a concept is "only" a backend abstracting the network and event layers and which offers API's for a frontend. On top of Downstage a frontend like a UpStage server can be built. And this is unspecific to a programming language. So UpStage is more like a application and Downstage is the core framework. At least that is the idea ("Separation of Concerns")...

I hope this outline makes more clear what the architecture is about.

Cheers,

Martin

# References

A Better Understanding of MVC (Model-View-Controller) Thanks To Steven Neiland. (Ben Nadel) 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>

DownStage. (2012). Retrieved September 28, 2015, from Foobarlab: <https://trac.foobarlab.net/downstage>

*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)