Cloud Application Development Project – T00158237 – Daniel Jameson

Using the Strangulation Design Pattern to Refactor a PHP application up to Cloud Hosting Services.

1. **Abstract**

Cloud Application development is a field that is becoming more relevant by the day. It is considered advantageous to a company or development team to be able to deploy an application to a hosting service, and this report will detail the steps involved in migrating an application to a hosting service such as Amazon Web Services.

In particular, it is going to use the Strangulation Design Pattern to research and implement a Microservice based architecture where a monolith was previously used.

1. **Introduction**

The code for the following application can be found at the following Github address: <https://github.com/DogPope/CloudPhpApp.git>

This report is going to look at the gradual migration of an application to the Amazon Web Services (AWS) platform, paying particular attention to platforms like Elastic Beanstalk. Applications can be gradually migrated over time to a cloud hosted service such as a relational database service (RDS) instance that ensures more safe storage of user data.

Cloud Computing will be utilised by moving a locally hosted application to online servers. This process allows potential developers to move the services to off-site locations, allowing for lower running overheads, and increased security in the form of custom Virtual Private Clouds (VPCs), security groups and Identity and Access Management (IAM) security privileges.

AWS has a range of support for a variety of languages and frameworks, facilitating whatever the company or individual developer requires. This report will assess a Hypertext Preprocessor (PHP) application that is to be moved to a cloud hosting service. It will go on to compare previous applications of the Strangulation Design Pattern and implement some of the recommendations provided by said applications.

Pertinent to this approach is the prioritizing of various services in terms of the need to refactor them. They will be assessed in terms of their importance to making a secure, easily configurable and modifiable application using the services available on the AWS platform. From there, the most important methods will be implemented, hopefully making the application better than it was before.

1. **Literature Review**

The Strangulation Design Pattern, otherwise known as the Strangler Fig Pattern, is a software design pattern that focuses on upgrading older systems into newer systems (docs.aws, 2025). This process can be done by assessing the application in various ways, such as database implementation, application logic and design patterns. This gives a team time to look at older more vulnerable parts of the application, such as secret storage and database credentials, gradually migrating them to a microservice-like pattern, to separate the services enough to work on each one individually.

The process can be initiated by assessing the application and assigning some type of ordering to it, defining which services are most and least essential to migrate. A database, for example, could be broken down into different functionality depending on the necessary use case, for example, with separate instances for customers, orders, etc. This would generally be regarded as a high-priority task for the strangulation pattern, as a problem with one aspect of a database means that the application cannot read and write to it. Breaking down into separate services for each major function would alleviate this possibility on a large scale, although for the purposes of this report, this particular step was not completed, as databases cost money to deploy.

*3.1 Case Studies*

*3.1.1 Green Button Project*

The Green Button is a sample of the pattern in practise. It was used to redesign a DataCustodian system belonging to that project (Yu Li, 2020). The process involved migrating the monolithic application to a microservice-based application communicating over a REST API. The benefits to this approach were that the application became more granularized than previous iterations, allowing the developers to change one area of a system without affecting other service implementations in the program.

The main goal of this project was migrating to a Domain Driven Design (DDD) from a monolithic application, and the report concluded the following about the end result: “microservice has high reliability and effectively improves the performance of other unmigrated modules” and it also suggests more incremental improvements going forward, such as the application of the sidecar design pattern, using something like Docker to implement Metrics services, etc for the main application. This would improve the information gathering service, and provide more data to draw conclusions from.

This application received benefits in the form of increased performance, through allowing multiple services to operate at the same time, in some way allowing multiple processes to run concurrently. The services were weighed up and assessed in the form of how beneficial it would be to transition each service to a different architecture.

*3.1.2 Unnamed Legacy Subsystem*

The authors Alessio Ferri and Tom Coggrave were responsible for migrating a legacy system in the United Kingdom(UK) to a cloud hosted microservice system (Ferri, 2024). The system described here was built over the course of 40 years and spanned roughly 7 million lines of code.

The old system was causing problems in that it was overly laborious and expensive to change, as changes made had to consider the full scale of the application to make changes, as a change in one area often affected something else that was coupled with the same functionality. The application also was very heavy on overheads, costing the company in question too much money, in their opinion.

An example of the work done on this project is the segmentation of data through displacing the legacy implementation of the data segmentations through changing an IBM BD2 instance to a more flexible PostgreSQL and Kafka solution.

1. **Methodology – Old Application Assessment**

The subject of this report will be a PHP application that was developed for a second-year project on Server-Side Development. It is a simple Create Update Delete (CRUD) application that facilitates users creating an account and allowing for the purchase of games through the application. It is connected to a MariaDB database, with tables for each relevant field. This report will go on to discuss the suitability of the Strangler Fig Pattern to migrate the application to AWS services instead of running locally.

*4.1 Security*

The application has practically non-existent security, storing the database connection information in Prepared Data Objects(PDO)s in individual files where they are connecting to the database. An updated application would refactor this information to be stored in a more secure location, such as with Amazon Secrets Manager.

In Amazon Secrets Manager, the application data for accessing the database can be stored separately from the application code, allowing for increased security, as the credentials are no longer in the application, and can only be accessed by individuals with the required IAM permissions to do so. This has increased the applications security for very little time expenditure, only requiring a few extra minutes to do after setting up the environment for the application.

*4.2 Database Refactoring*

The database here can be changed to allow for less fields. Several fields were removed altogether, as they were not necessary for their use case, such as taking in the persons real name when creating an account. The new refactored application takes in a Username field instead, instead of having the real data associated with the persons first and last names. This has also reduced overhead, as less characters need to be stored in the database and should cost less to host the service, as less data now has to go into the database.

All that is really necessary to know to ship any given product is the Eir Code associated with their address, as the information for the other address fields can be gotten elsewhere.

*4.3 General Application Structure*

The application code can be improved in many ways, starting with the directory structure, as it followed a nonsensical pattern, being grouped into sub-categories. The obvious solution to this would be to use the industry standard Model View Controller pattern, having main categorical directories based on their functions, rather than the original layout.

The original layout does not keep anything loosely coupled for reuse, and repeats a lot of code that might be refactored into other areas, where it could only be coded once.

*4.4 Framework Application*

Realistically, migration to a framework would not be suitable for this project, as even with as much separation as possible, this would require the main application to be re-written to accommodate it. The Laravel and Symfony Frameworks were considered for this purpose, but the layout of the application could not be closely aligned enough to make this approach work.

The necessary changes are too numerous considering the benefits of taking such an approach. All relevant entities would have to be refactored into classes instead of standalone programs, and the HTML would all have to be changed into a different language, blade.php for Laravel, and html.twig for Symfony. At that point, it could be argued that the original application no longer exists.

It was unilaterally decided that this kind of change would take too long to implement, and defeat the purpose of using the Strangler Fig pattern, therefore, this idea was abandoned. Maybe this idea will be considered for another application or project going forward.

1. **Methodology – Infrastructure Creation**

*5.1 VPC Creation*

Deploying this application to AWS involves first setting up the underlying infrastructure to accommodate the purposes of the application. The VPC is the first thing to set up for any project, allowing the developers to design and maintain the networking for the application. The benefits of an approach like this are that scalability is easily achieved, and Route Tables with Subnets that facilitate lots of different endpoints for many different services can be configured. This approach gives administrative users more control over how the application is deployed, supporting fine-grained control over services such as access to different parts of the application and setting inbound and outbound rules for security groups, which can then be applied to instances of those service types.

The architecture of the VPC can be found below:

A screenshot of a computer

AI-generated content may be incorrect.

Figure : Application VPC Configuration

*5.2 RDS Database Creation*

The creation of a database is a simple enough process. To refactor the security aspects described in the previous section, the database must have certain AWS Secrets applied to them. This project will have credentials supplied to it from Amazon Secrets Manager directly, to preserve the security of the system.

Amazon Secrets Manager in this case will be used to hold the database credentials, such as the username and password, keeping them in safe environments that cannot be accessed by outside entities.

This application uses a MariaDB database instance, and no real changes need to be made to it to facilitate cloud deployment. For the purposes of migrating to a Microservice Architecture, the correct way to do this would be to give a separate instance of database for each service, but this would be prohibitively expensive, and this will not be done for this project.

*5.3 Amazon Secrets Manager*

Central to database security in this application is the configuration of secrets on the Amazon side of the application. Needing the AWS SDK, importing the secret manager allows the application to access important database information. This allows holding important environment variables separate from the main application logic, and prevents the need for code reuse and greatly enhances application security. In the database file, the following script was added to retrieve the secrets held on Secrets Manager: A screen shot of a computer code

AI-generated content may be incorrect.

Figure : Amazon Secrets Manager Implementation

*5.4 IAM Roles*

Any good AWS project needs users and services that can interact with your code to varying effects. This needs to be configured via Identity and Access Management(IAM) roles, where each service needs the correct permissions to access their relevant services. In this case, any EC2 instance created by the Elastic Beanstalk service needs to have permission to talk to the database or between the frontend and backend subnets, where each service are stored.

*5.5 Elastic Beanstalk*

Amazons Elastic Beanstalk is a service that simplifies the application creation process by letting developers deploy to “environments” instead of a whole application upload, allowing for the easy deployment and changing of an environment. It allows for incremental changes to be made to an application by updating the environments where it is requested by the user. It does this by assigning version labels to the service. If a change is made, the version number needs to be changed to reflect the version changing.

This is where the process gets a bit more complex. Deploying any application to the service requires the underlying architecture described above. The default services could be used, but that would not give fine-grained control over the services described. First, an application needs to be made, to hold any environments the user might create. An application is just created with a name, and all it does is hold any environments associated with it.

Under the hood, this service creates additional resources that manage the applications services as appropriate. It configures load-balancers for the EC2 instances and Auto-Scaling groups as requested by asking for a “single instance” or a “highly available” environment, which is going to scale EC2 instances according to the requested availability. This report did not use a pre-configured database, as the creation of the database was done separately. The only major change from the textbook setup is the configuring of environment properties pertaining to the database name, endpoint and the secrets set up with Secret Manager.

Properly leveraging an Elastic Beanstalk environment is an easily managed process. The primary factor changing how environments operate is load balancing. An individual environment can have instances created for it that enhance application availability. The “Main” environment for this application was set up to have two load balanced instances set up, with a minimum availability of one, and a maximum of two load balanced instances across two public subnets, with the main application being held on two private subnets for the backend of the application.

1. **Methodology – Strangulation Pattern Application**

*6.1 Microservice Migration Overview*

The first major concern with moving to a Microservice Architecture should be granularity between services, allowing for different functionality between services. For example, the services “Customers” and “Games” could be moved to different areas, enhancing the sanity of the directory structure. The original layout was inherently insane, and had odd layouts that were very unintuitive to developers, using services that were poorly grouped and defined. The old structure was broken down messily into ‘sub-categories’ instead of a more modern source directory broken into functionality. The problem and the solution are both pictured below, using NeoVims NERDTree plugin, with the previous application being held in the “AWSApp” directory, and the Microservice-oriented design being held in the “CloudPhpApp” folder. Note that AWS Elastic Beanstalk looks for an Index.php file as a starting point. In both cases, this links to the Header.html file within the HTML folder as the application starting point.

|  |  |
| --- | --- |
| Old Application | New Application |
| Figure : Directory Sanity Improvement |  |

*6.2 Introduction of a Dependency Manager*

The dependency manager “Composer” was added over the course of this applications development cycle, as this would leave greater possibility for expansion in the future. Composer is a Dependency Manager for PHP that facilitates the use of various ancillary libraries and functions. In this case, it was necessary to import the AWS Software Development Kit(SDK) for access to AWS services like Secrets Manager. These secrets need to be accessed by the application for safe and secure access to the database. It means that application secrets are inserted via the Secrets Manager on the cloud hosted side and are not stored anywhere in the application directly.

Autoload is also configured with the Composer Dependency Manager. This allows for various parts of the application to be treated as dependencies, allowing the database methods to be simplified and removed from the main application logic. It is set to the ‘src’ directory, allowing the classes within src to be accessed elsewhere, and is used to simplify the database logic.

*6.3 Reconfiguring the Application Entrypoint*

An Amazon Elastic Beanstalk environment naturally starts from an “index.php” start point by default. This application was not designed with this in mind, and could lead to confusion down the road if it was arrived at later. The insertion of an “index.php” file allows for the entrypoint to be configured easily. From a user perspective, this application serves the index file which contains one line, a require statement that links to the “aboutus.php” file within the html directory. This About Us file contains basic information about the site, and more importantly, a link to the header.html file where the nav bar is stored. This facilitates easy application navigation by the user.

When the old application was run, Xampp could have been used directly to navigate directories, making sure something was always available to the end user. On a cloud environment, this is not the case, and a resulting 500 error will be thrown when trying to access pages that don’t exist. It was easier to add an index.php instead of routing to the endpoints separately.

*6.4 Database Method Reconfiguration*

Database management in the application should be broken down into two different areas, the database logic and the schema. The application could use its own implementation of the Database class, removing the need for repeated code in all the classes. The Schema was also simplified, aiming to reduce costs in the RDS database implementation for AWS. The “firstname” and “lastname” columns, for example, were simplified into “username” instead, and simple changes like this could reduce running costs and reduce the amount of potential failure points for the application.

The games database table was also simplified, going from ten columns to seven, and as a result, no longer holds redundant data such as the game description.

The schema is held in its own service, the “Core/SchemaManager.php” file. This communicates the schema to the RDS database, allowing for the creation of tables in an empty database, ensuring that the application development team can use it regardless of whether table definitions already exist on the database. This approach also gives the benefit of only creating an “order\_item” dependency where an “order” already exists, ensuring greater adherence to referential integrity.

The database methods were refactored into the “Core/Database.php” file. This file ensures that database connections are only accessed from one location, and need to be initialized once, adhering better to the Single Responsibility design principle.

This approach also makes it easier to expand the application, with only an import statement being required to access the database methods in the Database file.

Here is a comparison of the old approach and the new approach for adding a customer to the database: A black background with many small colored lights

AI-generated content may be incorrect.

Figure : Bind Values in File

This was refactored into a more simplified version in the new application like so: A screen shot of a computer program

AI-generated content may be incorrect.

Figure : Cleaner Database Implementation

As can be seen, the simplified version is much easier to parse and build on in future iterations of the application. This code is the equivalent of the above code, although the PDO object creation is above this, so may not be an exact comparison. Also note the added error checking, informing the user of whether the account was created successfully or not. The actions performed are the same, but the new version is cleaner.

*6.5 Versioning with Elastic Beanstalk*

Careful attention needs to be paid to this process, as when working with a Windows machine, the file upload process is complicated. Because servers online are generally a Linux server environment, the file separators are different than on the Windows Operating System (OS). Windows stores file separators as a backslash, unlike all other OS. Linux would not be able to parse differences in files, as it uses forward slashes instead, which is the more natural approach to take with file separators. Readers may be under the impression that this does not matter to a Web Application, given that the standard when being rendered in any browser that may access it is forward slashes, but alas. The issue is that Linux unzips the files expecting to find a forward slash as a path separator, instead of a backslash. There are two possible solutions to this problem.

*6.5.1 Changing the Zip Tool*

The use of the 7-Zip file archiving tool can circumvent this issue, as it zips folders with forward path separators by default. This would be a simple approach to the problem, as it involves running just one command, and it is the following: ‘& 'C:\Program Files\7-zip\7z.exe' a -tzip deploy.zip src\_directory/\* -r -mx=9’. This command would insert the files into a zipped “Deploy.zip” archive, with forward path separation.

*6.5.2 Environment Configuration*

The environment can also be configured to implement the same functionality. The installation of the “EB” Python package allows for the use of the AWS SDK to be used in conjunction with the project. Pythons package manager, pip, can be used to install the AWS EB Command Line Interface, allowing for the direct interaction with the environment within the project. From here, an environment can be made for the project, allowing for deployment on a programmatic level. This makes deployment of the application as simple as running one command, “eb deploy”.

A new environment can be configured, or an existing one can be configured in the files. In this case, because the setup for the AWS environment is rather complex, config files should be added for the VPC already created, with careful attention being paid to the Load-Balanced instance subnets, and the main instance subnets, as Elastic Beanstalk will try to create these with the wrong subnets by default.

For this reason, it was necessary, using the “eb deploy” approach, to create the environment configuration after creating the environment on AWS, then transferring the variables to the configuration files. After this setup is configured, deploying the application is as simple as running the command.

Elastic Beanstalk allows for incremental application changes, in addition to whatever type of version control the developers might decide to use. In the case of this application, a separate environment is available for each major iteration of the application. The first environment holds the basic application that does not access the database, and basically is just HTML and CSS at that point. The second environment is where the interesting stuff happens. The second environment holds the updated database with the Schema Manager and separation logic. This report went through several iterations of environments for different purposes, some of which can be seen below. A screenshot of a phone

AI-generated content may be incorrect.

Figure : Elastic Beanstalk Facilitates Multiple Environments

At this point in time, the gruntwork is mostly done, and the remaining application environment is the finished version with the necessary refactoring done to it.

1. **Results and Conclusions**

*7.1 Results*

The steps taken above have led to an application that is more robust and user friendly to users and developers alike. The code for the credentials is no longer stored insecurely, now relying on the Amazon Secrets Manager for this, as the results now get parsed from JSON in the environment instead of being inserted into the code directly.

The application is much more extensible and secure than its previous iteration. AWS helped facilitate this via the Elastic Beanstalk service and allowing with the construction of other services for the application, such as a VPC for networking and internet gateways, this application is now freely available to talk to the rest of the region that it is installed in. The abstraction provided by Elastic Beanstalk has simplified application deployment, creating the necessary resources for deployment with a single command, with changes implemented very quickly depending on the scope of the change.

Being configured across numerous Availability Zones has led to the stability of the application being much improved and should allow access to different areas and improve availability to more potential users. Any application set up in this manner should expect an increase in sales via more market access.

*7.2 Conclusions*

In conclusion, the application taken from its starting point here has metamorphosized into an application that should be accessible to all users, across a more configurable and adaptable setting. The suggestions from the application section could eventually be taken on board, in later versions of the application if any are created in future.

The goals set out here have largely been reached, although room for improvement still remains, namely in the framework upgrade and an internal gateway for the API to access the services.

*References*

docs.aws. (2025, March 20). *The Strangler Fig Pattern.* Retrieved from Amazon Web Services: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-aspnet-web-services/fig-pattern.html#:~:text=The%20strangler%20fig%20pattern%20was,tree%27s%20branches%20and%20overtakes%20it.

Ferri, C. (2024, April 10). *Martin Fowler Blog.* Retrieved from Uncovering the Seams in Mainframes for Incremental Modernisation: https://martinfowler.com/articles/uncovering-mainframe-seams.html

Yu Li, M. L. (2020, December 20). *Research Gate.* Retrieved from Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System: https://www.researchgate.net/publication/349568403\_Microservice\_Migration\_Using\_Strangler\_Fig\_Pattern\_A\_Case\_Study\_on\_the\_Green\_Button\_System