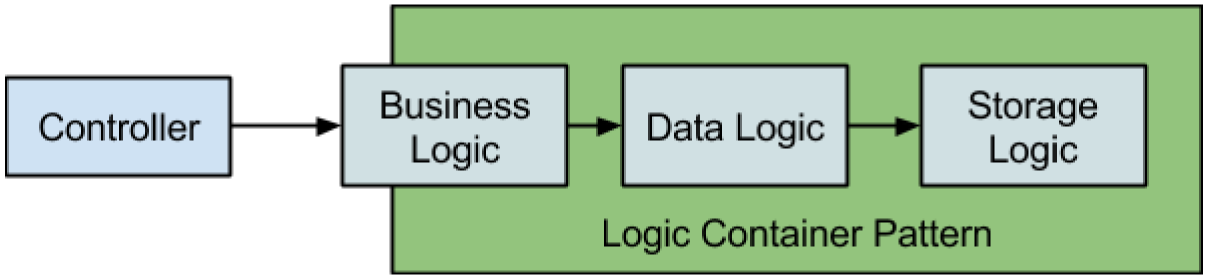
LOGIC CONTAINER PATTERN

Even when using patterns of Domain-Driven-Design, we're not really implementing a real world logic in our applications. Our classes still lack understanding of the business, they are still tightly coupled in their own independent, unrealistic logic flows and most importantly to each other. Most design patterns give independence to an extent; however, applying a change in the business flow requires dealing with interclasses logic. Having classes which `know` about each other does not help us achieve a standalone class cluster. Logic Container Pattern (LCP) takes each entity in a business flow and cuts them into independent layers.

In LCP, a business entity splits into 3 layers: Business Logic, Data Logic, Storage Logic. These 3 layers make use of each other consecutively and do not know anything how the other one acts except for public input and outputs. Let's get more practical: If your application follows MVC pattern, `M` is where LCP gets in. Your controllers only talk to Business Logic and have no idea what kind of relations the Business Logic have. For controller, Business Logic is an API entry point and its only reference in the business flow. Business Logic then talks to Data Logic elements or other Business Logic elements, and a Data Logic element talks only to other Data Logic elements or its own Storage Logic element. And a Storage Logic element cannot talk any other element.



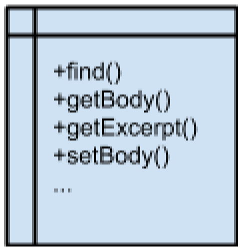
Let's define each element to make things clear.

Storage Logic: Storage Logic is probably the most familiar element in this pattern. It is actually a model, as in MVC. Its responsibility is to store data and provide logical, reasonable methods to get data. It is the lowest level element in LCP and contains data storage related logic flows. It saves, fetches, removes and presents a part of data. Below is a Storage Logic (read as Model) example:

```
<?php namespace LCP\Storage;

class Article
{
        protected $id;
        protected $title;
        protected $authorID;
```

```
            public static function find($id)
            {
                    $query = $dbh->prepare("SELECT * FROM articles WHERE id = :id");
                    $query->bindParam(':id', $id);
                    $query->execute();
                    $result = $query->fetch();

                    $book = new static();
                    $book->setId($result['id']);
                    $book->setTitle($result['title']);
                    $book->setAuthorID($result['author_id']);
                    $book->setBody($result[article_body]);

                    return $book;
            }

            public function setId() {}
            public function setTitle() {}
            public function setAuthorID() {}
            public function setBody() {}
            public function getId() {}
            public function getTitle() {}
            public function getAuthorID() {}
            public function getBody() {}
            public function getExcerpt($length = 100)
            {
                    // Returns first 100 characters of the body.
                    return substr($this->getBody(), 0, $length);
            }
    }
```

This is a glimpse of a Storage Logic for articles. It currently has several public methods, one of which is seen here as find(). It returns the found Storage\Article.



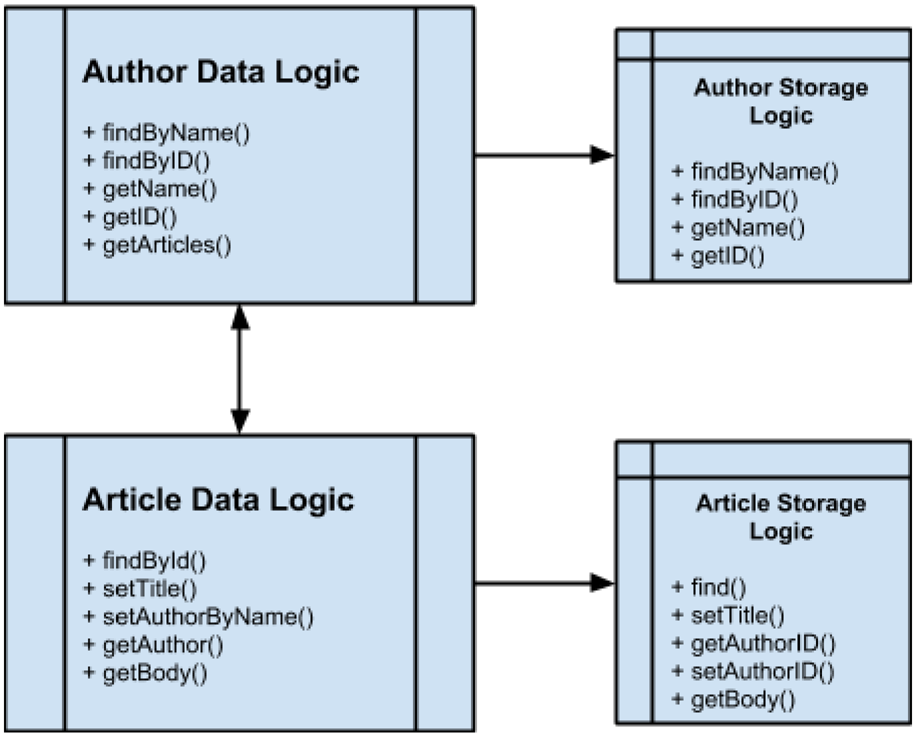
LCP tries to contain everything in a logical pattern and flow. An example of this is seen in method getExcerpt(). The database table does not have a field for excerpt and thus, its model lacks the excerpt property. However, it does have a getExcerpt method which returns the first 100 characters of the article body. An article is an independent, singular entity. Its title, body, number of pages, word count etc. are its sole and core properties which are not and cannot be independent entities on their own, logically. We don't think the word count of an article is a completely separate existence with its own logic. An excerpt, also, is a natural part of an article. However, on this example, we don't want to store it in our database and provide a method to accompany our Article. Later when we want to store excerpts with our articles, we will only change this specific and single method and get done. So we put a getExcerpt method here, because excerpts are natural and logical parts of articles.

Let’s go on with an Article’s data logic.

Data Logic: Data Logic is where we play with raw data from Storage Logic. We use the data we get from our Data Logic to put a logical flow in it and create a whole data logic. Data Logic elements can only talk to its own Storage Logic or other Data Logics. So you can only talk to your descent or your same level friends. A Data Logic cannot talk to other Data Logics’ descents, meaning Storage Logics. Let’s go on with the Article.

```
<?php namespace LCP\Data;

use LCP\Storage\Article as ArticleStorage;
use LCP\Data\Author as AuthorData;

class Article
{
	protected $currentArticle;

	public static function findById($id)
	{
		$article = ArticleStorage::find($id);
		$instance = new static();
		$instance->currentArticle = $article;

		return $instance;
	}

	public function setTitle($title) {
		return $this->currentArticle->setTitle($title);
	}
	public function setAuthorByName($authorName) {
		// Here we are talking to Author Data Logic, and not its Storage
Logic.
		$author = AuthorData::findByName($authorName);

		// Update the Author ID in Article Storage Logic. We are allowed to
		// talk to our own Storage Logic.
		return $this->currentArticle->setAuthorID($author->getId());
	}
	public function getAuthor() {
		return AuthorData::findById($this->currentArticle->getAuthorID());
	}
	public function getBody() {
		return $this->currentArticle->getBody();
	}
	public function setBody() {}
	public function getId() {}
	public function getTitle() {}


}
```

This is a draft Data Logic for articles, which are logical, practical entities in real world. It provides a static method to return a self instance with the data retrieved via Storage Logic of article. It provides several other public methods, like getAuthor. This is where our game with Data relations come to play. An article, as we said above, is an independent entity with its dependent properties such as title, excerpt, word count (because they are definitely dependent and minimal properties of an article)... However, an article would have an author. An author is, clearly, an independent entity in real world. Because of that, it has its own Data Logic. If a piece in our application has a Data Logic, everybody must use it. That means it is logically a separate entity and so it has its own Data Logic.



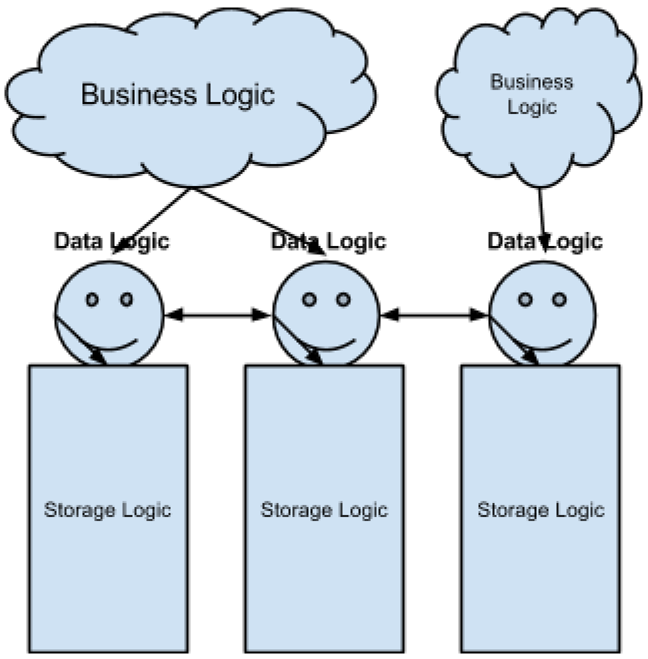
getAuthor() method in our Article Data Logic will return an instance of Author Data Logic. We don't return author ID as stored in Article Storage Logic, because it doesn't really mean anything to the outer world. Data Logics can talk to each other or to its Storage Logics.

Let's get to another example here. We usually have Users in our applications. Users usually have e-mails. We can store e-mails in various ways, but for now, let's assume we are using an e-mail table and a polymorphic e-mail relations table. E-mail is a standalone, single logical entity. How an e-mail is stored is not the business of outer world. In our design, we will have 1 E-mail Data Logic and 2 E-mail Storage Logics, one which will keep e-mail addresses and the other one will keep their relations. I'm not putting Storage Logics here, but you can guess what they look like by checking the Data Logic.

```php
<?php namespace LCP\Data;

use LCP\Storage\Email as EmailStorage;
use LCP\Storage\EmailRelation as EmailRelationStorage;

class Email
{
	protected $currentEmail;
	protected $currentRelation;

	public static function findByUserId($userId) {
		$relation = EmailRelationStorage::findByUserId($userId);
		$email = EmailStorage::findByID($relation->getEmailID());

		$instance = new static();
		$instance->currentEmail = $email;
		$instance->currentRelation = $relation;

		return $instance;
	}

	public static function insertForUser($email, $userId) {
		$email = EmailStorage::insert($email);
		$relation = EmailRelationStorage::insertForUser($email->getId(),
$userId);

		$instance = new static();
		$instance->currentEmail = $email;
		$instance->currentRelation = $relation;

		return $instance;
	}

	public function getEmailAddress() {
		return $this->currentEmail->getAddress();
	}

	public function getUserId() {
		if($this->thisRelation->getRightHandType() == ‘user’) {
			return $this->thisRelation->getUserId();
		} else {
			throw new Exception(‘This e-mail does not belong to a
	user.’);
		}
	}
	public function getCompanyId() {
		if($this->thisRelation->getRightHandType() == ‘company’) {
			return $this->thisRelation->getCompanyId();
		} else {
			throw new Exception(‘This e-mail does not belong to a
	company.’);
		}
	}
}
```

This Data Logic combines two storage logics and creates a single E-mail Data Logic. If we one day decide not to use a polymorphic email relations, all we have to do is to remove or update EmailRelationStorage and perhaps put our new logic in Email Data Logic. It's all about how much change you want to do; the effect of a change is determined by its logical shift.

So in LCP, we have observed 2 categories so far. Storage Logic, which is similar to our current understanding of Models and which is our data storage layer. It does have a logic within, however it does not extend to interlogical sphere, it has a single logic and deals only with it. Data Logic, however, is the second and latest step in the data flow of our application. It is the final data manipulator, accessor, provider for the outer world. It talks to other Data Logic elements and its Storage Logics. Let's imagine you are a Data Logic element. The data mass you are dealing with is the foods, e.g. water and bread. You, as the Data Logic, take water and bread from the outer world and send it to your Storage Logic, to your digestion system. Your digestion system is an independent entity and you have no idea how the food you sent to it is stored. It may be burnt into calories or stored as fat, but you don't care about it. You care about sending the food and receiving a signal about this action. You, as the Data Logic, can talk to other Data Logics, other people. You cannot talk to their digestion system, but you can interact with your own system. You can give good to other people and take food from them, Data Logics are able to talk to each other.



But, what is Business Logic doing around here? What is its purpose?

Business Logic: Business Logic can be thought of as an API. For a Storage Logic element, the outer world is Data Logic elements. For Data Logic elements, the outer world is Business Logic elements. And finally the outer world for Business Logic elements… is anything. But practically, it's probably the controllers of your applications. Business Logic elements provide an internal API for your whole application. They are not only playing with data, but also do any other action unrelated to data. They can move files, send e-mails, use Data Logics, talk to external web services, like Facebook, have configuration files for business rules, make use of localization, have helper libraries and even views for e-mails (because e-mail templates are usually application-independent, and thus controller independent).

For example, in my current development environment, I have a Client Business Logic, which provides me an interface to interact with client data and do e-mailings. I have another Business Logic for Projects and it undertakes file input sent from controllers, secures them, prepares them for further business flow, interacts with Project Data Logic. None of my controllers knows about the e-mailings, or file securing or editing them.

Business Logic elements have logical endpoints provided to outer world. When a new user registers on our website, we carry this to our User Business Logic. It creates the user via User Data Logic and sends a welcome e-mail. Controller does not command User Business Logic to send an e-mail or create the user via Data Logic. Business Logic decides all of the actions needs to be taken during a user registration. It can send a confirmation e-mail or a welcome e-mail and Controller would have no idea what has just happened. Controller is just a request gateway, specifically a HTTP request transmitter if the platform is on web.

For the outer world, Business Logic is the only thing they know. To make things easier, I usually extend Data Logic in a Business Logic. This gives me a comfort zone by providing already present methods quickly and I build additional business logics on top of Data Logic flow.

```php
<?php namespace LCP\Business;

use LCP\Business\Email as EmailBusiness;
use LCP\Business\Payment as PaymentBusiness;
use LCP\Business\User as UserBusiness;
use LCP\Data\Cart as CartData;
use LCP\Data\Billing as BillingData;

class Cart extends CartData
{
        /**
         * CartData Data Logic class would have required public methods,
         * such as newCart(), addToCart(), getTotal(), getUser() etc... These are
         * all data-related flows, so they are contained in CartData.
         *
         * public static function newCart() {}
         * public function addToCart() {}
         * public function getTotal() {}
         * public function getUser() {}
         * ...
         */

        public function getUser() {
                return UserBusiness::findByID($this->getUserID());
        }
        public function checkout(Array $paymentInfo, Array $billingInfo) {
                try {
                        $payment = PaymentBusiness::getPayment(
                                $paymentInfo['cc_no'],
                                $paymentInfo['ccv'],
                                $paymentInfo['cc_holder'],
                                $paymentInfo['cc_exp'],
                                $this->getTotal()
                        );

                        $billing = BillingData::newInvoice($billingInfo);

                        $this->getUser()->sendThankYouForShopping();

                        EmailBusiness::notifySalesMen($billingInfo);

                        return true;
                } catch (\Exception $e) {
                        return false;
                }
        }
}
```

This is a simple Business Logic for a shopping cart. It's simple, but provides a general overview of LCP. A Cart is a Business Logic. It extends its own Data Logic, `LCP\Data\Cart` and builds additional flows on top of the data logic. Extending is usually useful, because much part of our flows are data-related. However, Cart Business Logic is not only dealing with data.

When the user wants to checkout her cart, the application controller collects required input from the user and sends it to Cart Business Logic. Then Cart first processes the payment. Payment here is composed as another Business Logic, because it will deal with external payment processors, payment data logic (storing transaction details), do fraud tests etc. Then we create the invoice for current checkout transaction. Billing as seen above is not a separate Business Logic, but just a Data Logic. Billing will not have flows other than data manipulation, so it doesn't need a Business Logic. Data Logic is usually satisfactory with such elements. Then we send a Thank You notification to the user via User Business Logic. It is probably making use of Email Business Logic in itself, however a Thank You notification for a client is tightly bound to a user in logical terms, so we handle it in User Business Logic itself. I consciously didn't name that method as sendThankYouForShoppingEmail, because the user may want to get notifications via SMS and not E-mail. This decision is left solely to User Business Logic. And finally, we notify our sales men about this new purchase via email.

A summary for distinction between a Data Logic and Business Logic: if an entity is not dealing only with internal data, it's most probably a Business Logic.

Logic Container Pattern, as its name implies, tries to contain logical flows in their own entities and constitutes a comfort zone for logical relationships. LCP comes with several benefits:

1. *Truly independent logic*: Elements in the whole pattern are independent. They do have relationships, but these are logical relationships and if a logical shift occurs in the business, our elements will easily comfort it. Independence is achieved with containment, which lets each element to provide specific methods with specific input and outputs, and by not giving a single clue of what is going on inside the class and its methods. Each element provides logical entry points, they are artificially constructed flows; any change in the flow easily fits in the elements by making logical changes in them.
2. *Mobility*: What LCP provides is an internal API. If you build your LCP for your public facing web application, you can rapidly take your LCP and create a public API with it. Or you can move your code between platforms and languages, only by rewriting it in that language without spending time with planning. LCP is all about abstracting logics and standardizing input and output for these logics. I already completed a public facing API by moving my LCP that I used in web application. It was really, really fast to build. I only built routers, controllers, authenticators etc. and added API-related logic flows in the LCP; that was all it took.
3. *Centralization*: We don't usually have multiple entities for a single logical term. LCP gives you centralized logical mechanisms to achieve a whole flow. Any action logically related to a Client is handled in its, for example, Business Logic layer. It centralizes your application in a humane way.
4. *Substitution*: Especially for your tests, you can easily substitute your logic layers. Consider Data Logic, extended by a Business Logic. You can replace your Storage Logic in this Data Logic with a data faker and get done. Data Logic does not care how the data is retrieved or stored; but Storage Logic does.
5. *Ease of construction and manipulation*: As the whole pattern is based on humane logical flows, it's probably the easiest pattern you can build from scratch. You construct your LCP by talking to yourself and putting how you think the flow in the code.
6. *Communicative*: Its logical base makes LCP very communicative between disciplines. You can bring your class diagrams to a sales meeting and most people will be able to understand what you are talking about.

Oytun Tez, *2014-08-04*
oytun.co
oytun@motaword.com