Concordia University
Department of Computer Science
and Software Engineering

Software Architecture and Design II
SOEN 344 S --- 2017

Chronos
AOP Migration Technical Report

Team members	
Name	**Student ID**
An Ran Chen	27277385
Christiano Bianchet	27039573
Philippe Kuret	27392680
Alexander Rosser	27543069
Saif Mahabub	27392974
Amr Mourad	26795331
Youness Tawil	27013353
Adriel Fabella	27466005

Table of Contents

Pattern Migration to Aspect Oriented Programming

The new requirements demand that an architectural pattern be used in an aspect-oriented context. This leaves us with the option to implement a completely new pattern or simply take an already existing pattern and migrate it to an aspect-oriented context. Before selecting an option, the pre-existing patterns were analyzed.

Effort Analysis of the Migration

This whole migration process required a significant amount of reverse engineering as well as re-engineering. By considering what we were given, we had to understand the functionalities and then, examine it carefully to redesign some components so that we may build upon it. More precisely, the effort consisted of first analyzing the existing source code to find out the segments that implement the crosscutting functionality. Afterwards, we move on to the transformation of the existing code into an aspect-oriented reformulation. Thus, to fully take advantage of the potential benefits of the AOP style of programming there is definitely a need for migration support of existing systems.

Current Patterns

The system was originally designed using multiple architectural patterns which included the use of mappers, Identity maps, table data gateways and a unit of work. While reviewing the source code, it was observed that the unit of work pattern was not implemented correctly. We chose to migrate that pattern since there were many fixes which needed to be added to it and we therefore assumed we would already be familiar with it.

Advantages of AOP

In the original design of the system, the action of creating a new reservation from controller to UoW (Unit of Work) went as follows:

1- Reservation Controller → 2- Reservation Mapper → 3- Reservation UoW

Migrating the reservation UoW will remove the task of registering dirty objects from the reservation mapper, as the aspect will take over that task. This allows for a better separation of concerns. This separation makes for increased modularity as the mapper now takes care

of a more specific task as it no longer deals with the removal of object registration from the unit of work. Another advantage of migrating this pattern to an aspect-oriented design is the reduction of cross-cutting. Currently the task of tracking and adding dirty objects is split across multiple classes, as the mapper must register dirty objects and the UoW keeps track of these objects. Using aspects, the registration of dirty objects can be done by the UoW as it will be called upon during the process of object modification.

PHP AOP Framework

The selection of frameworks to implement the migration was filtered down to two selections: AOP and Go! Aspect-Oriented. After reviewing both frameworks, it was established that both function in a very similar fashion, the main differences were found in their installation and documentation. Go! Aspect-Oriented offered more documentation and tutorials online which will help improve the learning process for the team. The installation of AOP required pecl, a repository of PHP extensions. The installation of Go! Aspect-Oriented was evaluated to be much simpler as it relied on Composer, which was already installed on all our machines as it was needed to install the original Chronos project.

Installation Process

Once Composer is installed on the machine, downloading the framework simply consists of running the following line in bash:

```
$ composer require goaop/framework
```

The next step is to create and configure an Aspect kernel for our project. Once ready, aspects can be created and registered to the kernel. The full installation guide can be found at: https://github.com/goaop/framework

Migration Plan

The migration plan is divided into three sections:

1) Architecture pattern identification
2) Code segment identification
3) Refactor phase

The first phase consists of identifying the architectural pattern - the interceptor. The second phase identifies the code segments that are most appropriate for migration to aspects. It is where the source code is analyzed carefully to locate suitable aspects. Afterwards, during

the refactoring phase the code is transformed for the crosscutting concerns to be comprehended by separate aspects instead of the original classes.

~~Migration Plan~~ (refactored in the next section)

Once the unit of work pattern has been fixed, the migration process may begin. The migration will affect all files relating to the unit of work pattern. The project has a single unit of work class; ReservationUoW. ReservationUoW's task is to keep track of changes made to reservations by the ReservationMapper. Once the mapper is ready to commit, it returns the list of creatio ns, updates and deletion of Reservation instances. The migration of this unit of work class will entail changes to the ReservationMapper class. The mapper class will no longer take care of registering dirty objects to the UoW class. This includes registering the creation of new instances, the modification of existing instances and the deletion of instances. This allows for the minimization of tasks completed by the mapper and thus increasing modularity. Similarly, cross cutting is reduced as all tasks related to the registration of dirty objects is done within the aspects instead of between the UoW and mapper class.

The ReservationMapper's following methods will be affected:

-Create(...)

-set(...)

-delete(...)

-done(...)

Updated Migration plan

Commented [1]: To add other concerns

Commented [2]: what do you mean here?

After more in depth research of multiple php aop frameworks, it was concluded that the context passing required to implement the previous plan may not function as needed. For that reason, the pattern to migrate was changed. It was decided to implement the interceptor pattern. The benefits of using this pattern in aspect oriented context is similar to that explained above for the unit of work pattern. The pattern will be used to verify that a reservation being created or modified belongs to the user performing these actions. This will affect the ReservationMapper file in the following ways:

- showModifyForm(…)method will no longer authenticate owner. A new aspect will complete this task before execution of the method
- -modifyReservationForm(…)method will no longer authenticate owner. A new aspect will complete this task before execution of the method

1. Reservation Modification

The use of AOP will serve to intercept the *'modifyReservation'* method and check few requirements *before* the execution of the method. The requirements include: check whether the requested reservation exists and is owned by a user. This separates the requirement insurance checking process from the actual modification. That way, the crosscutting concern is better modularized since they are no longer tangled together. The aspect "ReservationAspect" will be created.

```php
/**
 * Method that will be called before modifyReservation method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->modifyReservation(*))")
 * @return \Illuminate\Http\Response
 */
public function beforeModifyReservation(MethodInvocation $invocation)
{
    $obj = $invocation->getThis();
    echo 'Calling Before Interceptor for method modifyReservation: ',
    is_object($obj) ? get_class($obj) : $obj;
    $reservationMapper = ReservationMapper::getInstance();
    // Find Reservation by id
    $reservation = $reservationMapper->find($invocation->getArguments()[1]);

    if ($reservation === null || $reservation->getUserId() !== Auth::id()) {
        return abort(404);
    }
    return null;
}
```

2. Reservation Creation - Show Request Form

When a user creates a reservation, the system first requests a *reservation making* form which is handled by the method *showRequestForm* in the *ReservationController*. The method was initially designed to open a reservation session for the user in order to handle reservation resource concurrency. However, in addition to this functionality, this method also checks a series of conditions to generate different scenarios that will either allow the reservation form request or deny it. There are several conditions under which the request will be denied:

1) If another student is reserving the same room on the same timeslot
2) If the current user has a reservation session underway
3) If the current user has reached the weekly limit of reservations
4) If the waitlist for the reservation is full
5) If the requested reservation is in the past

With the AOP migration, instead of calling the *showRequestForm* method directly, the *ReservationController* will use the *Interceptor pattern* to check all the constraints before the execution of the method. This change utilizes separation of concerns to provide better code

maintainability. This migration allows the base code to remain *oblivious* to this newly introduced aspect, allowing the functionality to be developed independently of it. The following code will be included in “ReservationAspect”.

Condition 1) Check if another student is reserving the same room on the same timeslot

```
/**
 * Method that will be called before showRequestForm method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->showRequestForm(*))")
 * @return \Illuminate\Http\Response
 */
public function beforeRequestFormCheckSession(MethodInvocation $invocation)
{
    $obj = $invocation->getThis();
    echo 'Calling Before Interceptor for method showRequestForm: ',
    is_object($obj) ? get_class($obj) : $obj;

    $roomName = $invocation->getArguments()[1];
    $timeslot = $invocation->getArguments()[2];

    $session = new ReservationSession(Auth::id(), $roomName, $timeslot);
    $sessionTDG = ReservationSessionTDG::getInstance();

    // Concurrency handling: lock the resource if another student is reserving.
    if($sessionTDG->checkLock($session)){
        return redirect()->route('calendar', ['date' => $timeslot->toDateString()])
            ->with('error', sprintf("Another student has a session underway. Please wait patiently
            or request for another room."));
    }
    return null;
}
```

Condition 2) Check if the current user has a reservation session underway

```
/**
 * Method that will be called before showRequestForm method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->showRequestForm(*))")
 * @return \Illuminate\Http\Response
 */
public function beforeRequestFormChekOwnSession(MethodInvocation $invocation)
{
    $obj = $invocation->getThis();
    echo 'Calling Before Interceptor for method showRequestForm: ',
    is_object($obj) ? get_class($obj) : $obj;

    $roomName = $invocation->getArguments()[1];
    $timeslot = $invocation->getArguments()[2];

    $session = new ReservationSession(Auth::id(), $roomName, $timeslot);
    $sessionTDG = ReservationSessionTDG::getInstance();

    // Concurrency handling: lock the other resources if the user is trying to access multiple
    // resources at the same time.
    if($sessionTDG->checkSessionInProgress($session)){
        return redirect()->route('calendar', ['date' => $timeslot->toDateString()])
            ->with('error', sprintf("You already have a session underway. Please complete
            it before you process to a new reservation."));
    }
    return null;
}
```

Condition 3) Check if the current user has reached the weekly limit of reservations

```php
/**
 * Method that will be called before showRequestForm method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->showRequestForm(*))")
 * @return \Illuminate\Http\Response
 */
public function beforeRequestFormCheckWeeklyLimit(MethodInvocation $invocation)
{
    $obj = $invocation->getThis();
    echo 'Calling Before Interceptor for method showRequestForm: ',
    is_object($obj) ? get_class($obj) : $obj;

    $timeslot = $invocation->getArguments()[2];
    $reservationMapper = ReservationMapper::getInstance();

    if ($this->reachedWeeklyLimit($reservationMapper, $timeslot)) {
        return redirect()->route('calendar', ['date' => $timeslot->toDateString()])
            ->with('error', sprintf("You've exceeded your reservation request limit (%d) for this week.",
                static::MAX_PER_USER_PER_WEEK));
    }

    return null;
}
```

Condition 4) Check if the waitlist for the reservation is full

```php
/**
 * Method that will be called before showRequestForm method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->showRequestForm(*))")
 * @return \Illuminate\Http\Response
 */
public function beforeRequestFormCheckWaitlistFull(MethodInvocation $invocation)
{
    $obj = $invocation->getThis();
    echo 'Calling Before Interceptor for method showRequestForm: ',
    is_object($obj) ? get_class($obj) : $obj;

    $reservationMapper = ReservationMapper::getInstance();
    $roomName = $invocation->getArguments()[1];
    $timeslot = $invocation->getArguments()[2];

    // check if waiting list for timeslot is full
    $reservations = $reservationMapper->findForTimeslot($roomName, $timeslot);

    if (count($reservations) >= static::MAX_PER_TIMESLOT) {
        return redirect()->route('calendar', ['date' => $timeslot->toDateString()])
            ->with('error', 'The waiting list for that time slot is full.');
    }

    return null;
}
```

Condition 5) Check if the requested reservation is in the past

```php
/**
 * Method that will be called before showRequestForm method
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->showRequestForm(*))")
 * @return \Illuminate\Http\Response
 */
public function beforeRequestFormCheckNotPast(MethodInvocation $invocation)
{
    $obj = $invocation->getThis();
    echo 'Calling Before Interceptor for method modifyReservation: ',
    is_object($obj) ? get_class($obj) : $obj;

    $timeslot = $invocation->getArguments()[2];
    $timeslot = Carbon::createFromFormat('Y-m-d\TH', $timeslot);

    // Prevent reserving in the past
    if ($timeslot->copy()->isPast()) {
        return redirect()->route('calendar', ['date' => $timeslot->toDateString()])
            ->with('error', 'You cannot reserve time slots in the past.');
    }

    return null;
}
```

Other Concerns to be Resolved Through AOP:

1. Development Mode Logging Concern:

Logging is not related to a particular class or application component, and involves constantly including the logging module and independently calling it over and over again within method definitions. An example of this is logging system events while in development mode, which are not to be visible in a production environment. By separating logging to an aspect-oriented implementation, the concern is treated completely separately of the application code and can be extracted simply by deactivating the aspect. This also results in cleaner code within the involved classes by not including code that is unrelated to the functioning of the system.

The following are instances, among other, where logging is used within methods in the ReservationController to keep track of system events around the reservation context:

```php
public function viewReservationList(Request $request)
{
    app('debugbar')->info("Entered method 'viewReservationList'");

    $reservationMapper = ReservationMapper::getInstance();
    $reservations = $reservationMapper->findPositionsForUser(Auth::id());

    return view('reservation.list', [
        'reservations' => $reservations,
    ]);
}
```

```php
public function requestReservation(Request $request, $roomName, $timeslot)
{
    app('debugbar')->info(
        "Entered method 'requestReservation'\n
         with parameters: \n"
        .
        json_encode(func_get_args())
    );
        ...
```

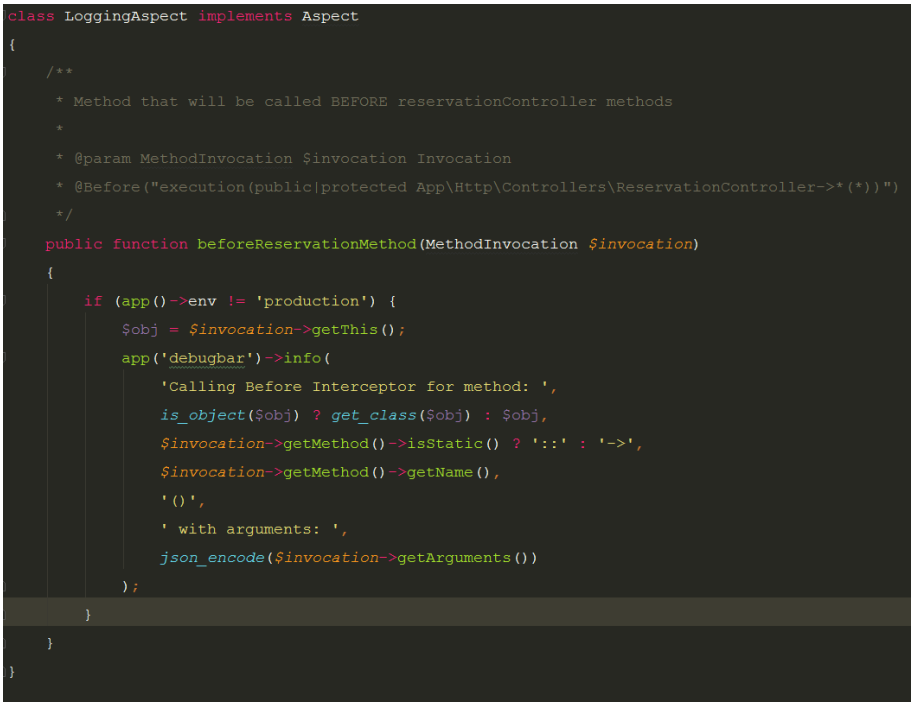
[*ReservationController* method definitions with logger calls]

And many more throughout the class.

Creating a *LoggerAspect* class removes the cross cutting of the logger interface in the involved classes/methods. In this case, the pointcut will be all the accessible methods in the *ReservationController* class:

```
@Before("execution(public|protected App\Http\Controllers\ReservationController->*(*))")
```

The advice is the definition that runs *before* these methods. Below is the implementation, which checks the whether or not the application is in production mode, and only logs the information if it's not.



```
class LoggingAspect implements Aspect
{
    /**
     * Method that will be called BEFORE reservationController methods
     *
     * @param MethodInvocation $invocation Invocation
     * @Before("execution(public|protected App\Http\Controllers\ReservationController->*(*))")
     */
    public function beforeReservationMethod(MethodInvocation $invocation)
    {
        if (app()->env != 'production') {
            $obj = $invocation->getThis();
            app('debugbar')->info(
                'Calling Before Interceptor for method: ',
                is_object($obj) ? get_class($obj) : $obj,
                $invocation->getMethod()->isStatic() ? '::' : '->',
                $invocation->getMethod()->getName(),
                '()',
                ' with arguments: ',
                json_encode($invocation->getArguments())
            );
        }
    }
}
```

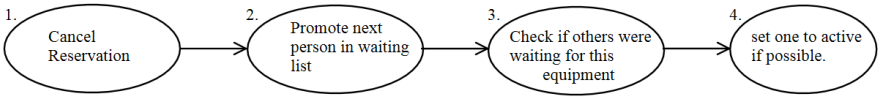
[logger aspect class]

This means that the original methods do not need to be aware or explicitly call any logger functions.

If certain methods need different logging structures than others, multiple definitions (advice) within the logging aspect can be added with different pointcuts referring to different, or a more specific, set of methods rather than having a single one as used in this example.

2. Reservation Cancellation - Equipment Add-On:

One of the advantages of using aspects is to treat the system as a blackbox and not modify current implementation. When equipments were added onto the system, it affected the waitlisting and created a dependency between reservations across multiple rooms. This means that if a reservation is waiting on another one in order to be active, it needs to be notified if that other reservation gets cancelled. Basically, the flow is as follows:



The only operation that actually involves the current user is the first one, namely the cancellation of the reservation. The rest are computation done for the overall system status. By using AOP, the '*cancelReservation'* method can be intercepted in order to do whatever needs to be done upon a cancellation. In this case, these other functionalities can be done ***after*** the successful completion of the method, meaning after cancellation of the reservation. As previously stated, this allows the underlying implementation to stay the same and separating it from the whole process of promoting other reservations due to equipment dependency. The following aspect will be created "*EquipmentAspect*" and can be extended to involve other equipment operations. The method *beforeCancellation* will intercept the parameters and store the reservation object **before** the method for future interactions, and the *afterCancelReservation* will do the necessary work **after** the method completes, namely steps 2, 3, and 4.

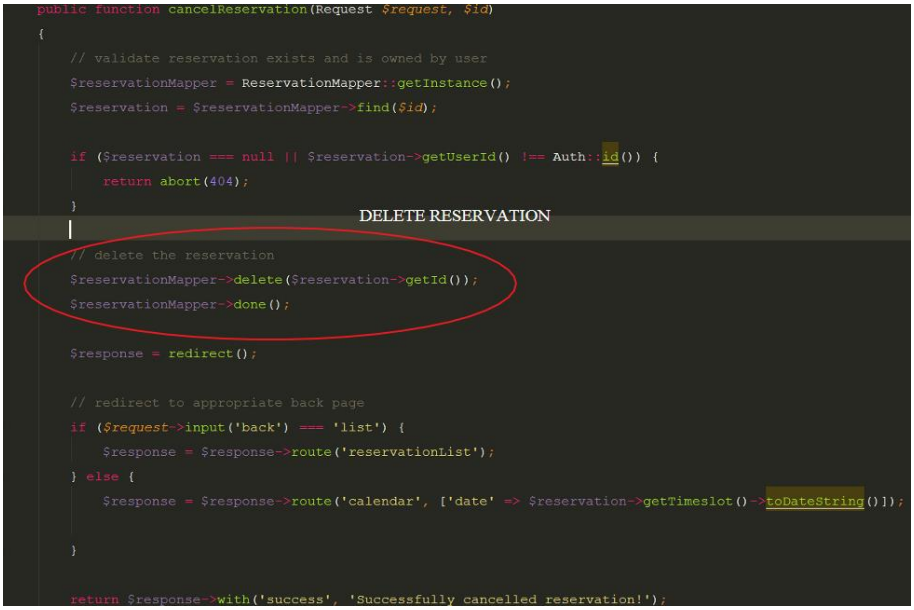
```php
/**
 * Method that will be called before cancelReservation.
 * Reservation object stored for manipulation post-cancellation.
 *
 * @param MethodInvocation $invocation Invocation
 * @Before("execution(public App\Http\Controllers\ReservationController->cancelReservation(*))")
 */
public function beforeCancellation(MethodInvocation $invocation)
{
    $arguments = $invocation->getArguments();
    $resId = $arguments[1];
    $reservationMapper = ReservationMapper::getInstance();
    $reservation = $reservationMapper->find($resId);
    static::$currentRes[$resId] = $reservation;
}
```

[beforeCancellation]

```
/**
 * Promote other reservations that are dependent on this one, if this one was active!
 *
 * @param MethodInvocation $invocation
 * @After("execution(public App\Http\Controllers\ReservationController->cancelReservation(*))")
 */
public function afterCancelReservation(MethodInvocation $invocation)
{
    $arguments = $invocation->getArguments();
    $resId = $arguments[1];
    $reservation = static::$currentRes[$resId];
    $reservationMapper = ReservationMapper::getInstance();

    $cancelledEquip = $reservation->getEquipmentId();
    $resTimeslot = $reservation->getTimeslot();

    /**
     * if reservation was active then need to replace it with next in waiting list,
     * and check if someone else needs to be promoted.
     */
    if (!$reservation->getWaitlisted()) {...}

    unset(static::$currentRes[$resId]);
}
```

[afterCancelReservation] (Reduced size for purpose of screenshot)

As a result, the original *cancelReservation* method behaves as it did before updating the system and is significantly smaller in scope. It only deletes the reservation.



[cancelReservation definition with the work delegated to the aspect]

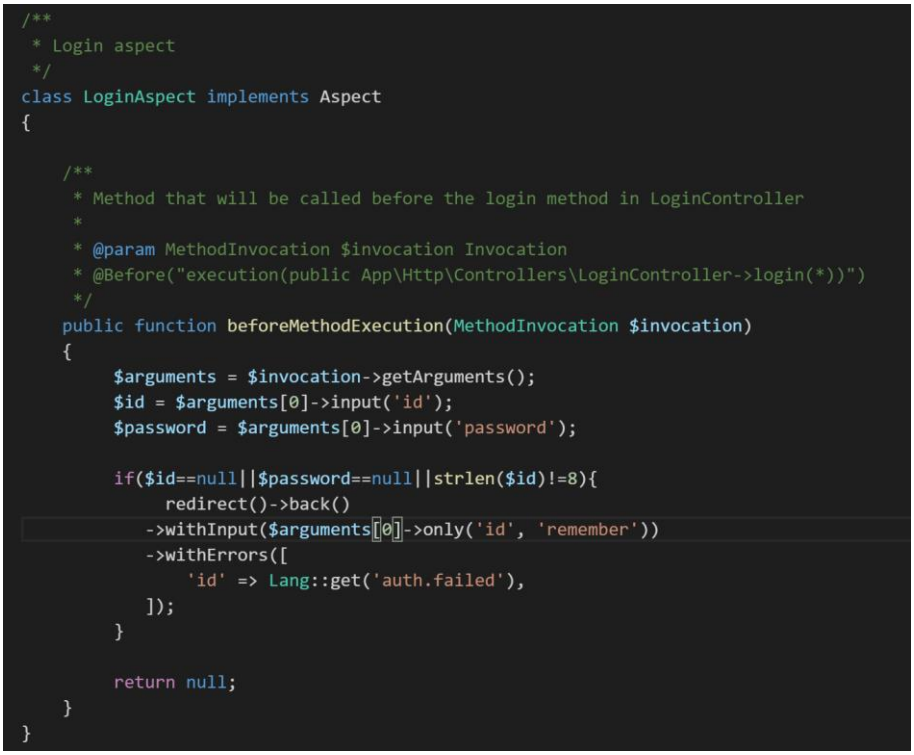
3. Calendar Controller – View Calendar

One simple yet effective use of Aspect Oriented Programming in conjunction with this application is to use an aspect to handle input validation on the calendar controller. When the view calendar method is called, the request parameter contains a date input. This input must be validated in multiple ways. Firstly, the input itself is validated on the front-end to ensure that the appropriate format is followed. Secondly, the date parameter must not contain a date which has already past. This check can be provided by an aspect using a pre-condition for the view calendar method. The precondition executes beforehand and checks if the date is in the past and if so, redirects the user back to the page displaying an appropriate error message. This removes the potential cross-cutting concerns in the view calendar method by separating validation from execution logic and placing the former in an aspect.

```php
/**
 * Calendar aspect
 */
class CalendarAspect implements Aspect
{

    /**
     * Method that will be called before the viewCalendar method in CalendarController
     *
     * @param MethodInvocation $invocation Invocation
     * @Before("execution(public App\Http\Controllers\CalendarController->viewCalendar(*))")
     */
    public function beforeMethodExecution(MethodInvocation $invocation)
    {
        $arguments = $invocation->getArguments();
        $date = $arguments[0]->input('date');
        $date = Carbon::createFromFormat('Y-m-d\TH',$date);

        if ($date->copy()->isPast()) {
            return redirect()->route('calendar',['date' => $date->toDateString()])
                    ->with('error', sprintf("You cannot view past reservations."));
        }

        return null;
    }
}
```

4. Login Controller - Login Validation

Another use for AOP is the separation of validation of user login credentials with the action of actually logging in. The validation of user credentials is a two-fold action. Firstly, the fields must be not empty and follow the format specified, and secondly they must correspond to credentials which exist within the database. In order to ensure that the LoginController does not take care of the trivial matter of ensuring that the fields are entered correctly, the LoginAspect will intercept the method call and ensure the format is respected before proceeding. While there may be some front-end checks to ensure the information is valid, adding this aspect provides further robustness to the system. The check uses a precondition on the login method in LoginController which executes before the method. If the id or password input parameters are null or if the id does not fit the 8 digit format, the login process will fail immediately, redirecting the user back to the login form and displaying the appropriate error. This demonstrates separation of cross-cutting concerns by having the aspect handle formatting validation and the controller handle database validation.



```php
/**
 * Login aspect
 */
class LoginAspect implements Aspect
{

    /**
     * Method that will be called before the login method in LoginController
     *
     * @param MethodInvocation $invocation Invocation
     * @Before("execution(public App\Http\Controllers\LoginController->login(*))")
     */
    public function beforeMethodExecution(MethodInvocation $invocation)
    {
        $arguments = $invocation->getArguments();
        $id = $arguments[0]->input('id');
        $password = $arguments[0]->input('password');

        if($id==null||$password==null||strlen($id)!=8){
            redirect()->back()
           ->withInput($arguments[0]->only('id', 'remember'))
           ->withErrors([
               'id' => Lang::get('auth.failed'),
           ]);
        }

        return null;
    }
}
```