МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ имени М.В.ЛОМОНОСОВА

Физический факультет

«Создание web-фреймворка для решение дифференциальных уравнений»

Курсовая работа по курсу “Основы web-технологий”

студентки 2 курса

Назаровой Е.А.

Преподаватель
Алексеев А.А.
«___»_____ 2016 г.

Москва-2016

Оглавление

Введение.

В данной работе моей задачей было создание программы, делающей решение дифференциальных уравнений для пользователей максимально удобным. Всю работу можно условно разделить на три части: написание кода на языке R для решения дифференциальных уравнений, написание программы на php с помощью php-фреймворка Laravel для непосредственного взаимодействия с пользователями и создание базы данных для хранения информации пользователей.

PHP

PHP (*Hypertext Preprocessor* — «PHP: препроцессор гипертекста»; первоначально *Personal Home Page Tools*[4] — «Инструменты для создания персональных веб-страниц») — скриптовый язык общего назначения, интенсивно применяемый для разработки веб-приложений. В настоящее время поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов.

Для работы с php-кодом я воспользовалась php-фреймворком Laravel.

Laravel — бесплатный веб-фреймворк с открытым кодом, предназначенный для разработки с использованием архитектурной модели MVC (англ. *Model View Controller* — модель-представление-контроллер).

Ключевые особенности, лежащие в основе архитектуры Laravel:

- *Пакеты* (англ. *packages*) — позволяют создавать и подключать модули в формате Composer к приложению на Laravel. Многие дополнительные возможности уже доступны в виде таких модулей.
- *Eloquent ORM* — реализация шаблона проектирования ActiveRecord на PHP. Позволяет строго определить отношения между объектами базы данных. Стандартный для Laravel построитель запросов Fluent поддерживается ядром Eloquent.
- *Логика приложения* — часть разрабатываемого приложения, объявленная либо при помощи контроллеров, либо маршрутов (функций-замыканий). Синтаксис объявлений похож на синтаксис, используемый в каркасе Sinatra.

- *Обратная маршрутизация* связывает между собой генерируемые приложением ссылки и маршруты, позволяя изменять последние с автоматическим обновлением связанных ссылок. При создании ссылок с помощью именованных маршрутов Laravel автоматически генерирует конечные URL.
- *REST-контроллеры* — дополнительный слой для разделения логики обработки GET- и POST-запросов HTTP.
- *Автозагрузка классов* — механизм автоматической загрузки классов PHP без необходимости подключать файлы их определений в *include*. Загрузка по требованию предотвращает загрузку ненужных компонентов; загружаются только те из них, которые действительно используются.
- *Составители представлений* (англ. *view composers*) — блоки кода, которые выполняются при генерации представления (шаблона).
- *Инверсия управления* (англ. *Inversion of Control*) — позволяет получать экземпляры объектов по принципу обратного управления. Также может использоваться для создания и получения объектов-одиночек (англ. *singleton*).
- *Миграции* — система управления версиями для баз данных. Позволяет связывать изменения в коде приложения с изменениями, которые требуется внести в структуру БД, что упрощает развёртывание и обновление приложения.
- *Модульное тестирование* (*юнит-тесты*) — играет очень большую роль в Laravel, который сам по себе содержит большое число тестов для предотвращения регрессий (ошибок вследствие обновления кода или исправления других ошибок).
- *Страничный вывод* (англ. *pagination*) — упрощает генерацию страниц, заменяя различные способы решения этой задачи единым механизмом, встроенным в Laravel.

R

R- мощный язык для статистических вычислений и графики, который может справиться поистине с любой задачей в области обработки данных. Он работает во всех важных операционных системах и поддерживает тысячи специализированных модулей и утилит. Все это делает R замечательным средством для извлечения полезной информации из гор сырых данных.

Система дифференциальных уравнений Лоренца.

Эта динамическая система была введена Лоренцем для описания идеализированного поведения земной атмосферы. Модель Лоренца описывает движение трех переменных X, Y и Z с начальными состояниями X(0)=Y(0)=Z(0)=1. a, b, c – параметры. Эта система возникла в задаче о моделировании конвективного течения жидкости, подогреваемой снизу. Такое течение описывается системой дифференциальных уравнений в частных производных.

$$\frac{dX}{dt} = a \cdot X + Y \cdot Z$$
$$\frac{dY}{dt} = b \cdot (Y - Z)$$
$$\frac{dZ}{dt} = -X \cdot Y + c \cdot Y - Z$$

Метод решения СДУ.

Для решения дифференциальных уравнений на языке R мы использовали пакет deSolve, предназначенный для таких задач с начальными значениями как :

- Обыкновенные дифференциальные уравнения
- Дифференциальные алгебраические уравнения
- Дифференциальные уравнения в частных производных

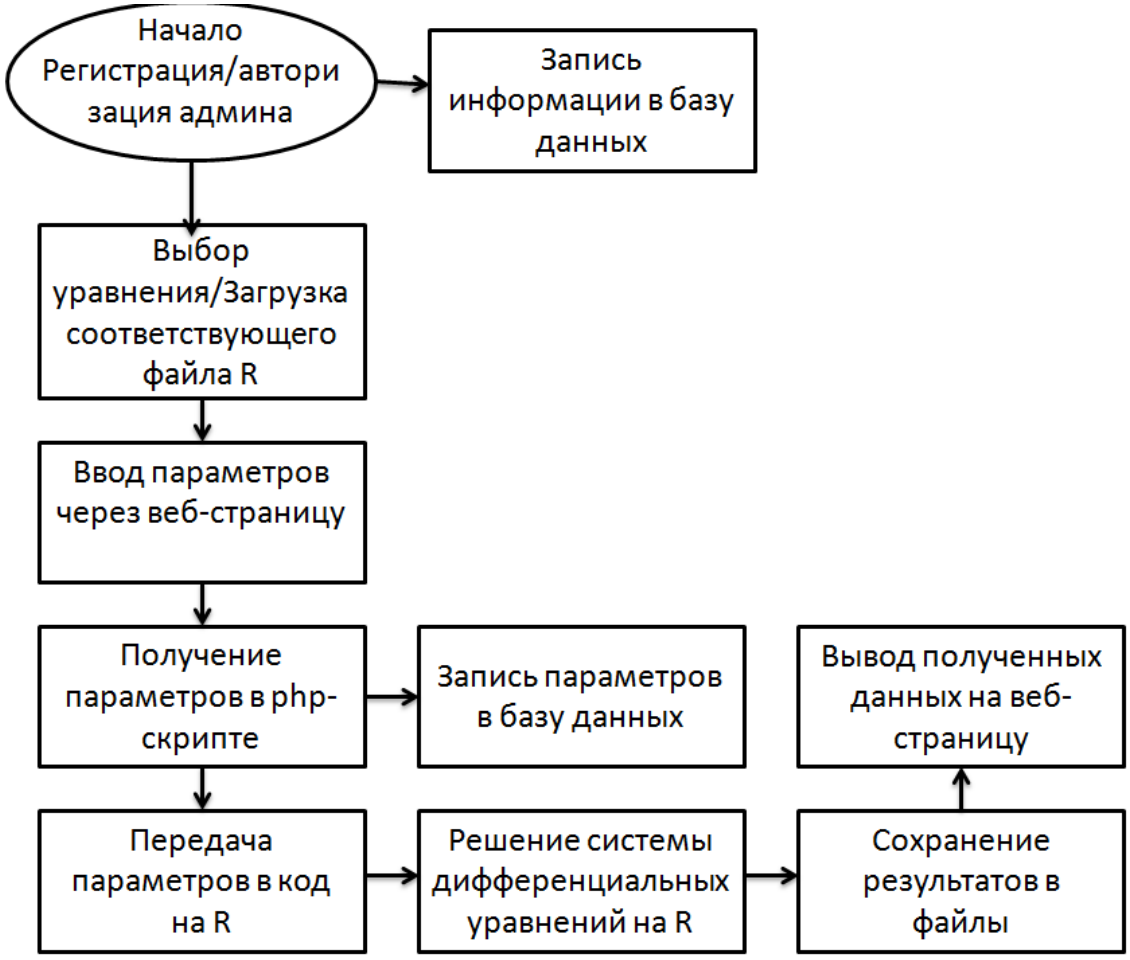
Он решает дифференциальные уравнения такими методами как метод дифференцирования назад, метод Адамса, метод Рунге-Кутта-Радо и некоторыми другими методами Рунге-Кутта.

Для решения нашей системы уравнений, мы используем метод ФДН(формула дифференцирования назад). В нем используется подход к построению численных методов для решения ОДУ, основанный на расширении шаблона разностной схемы. Полученные численные методы (первые схемы такого рода были получены Адамсом) носят название линейных многошаговых. Общий вид ФДН методов таков:

$$u_{n+1} = \sum_{j=1}^{k} a_j u_{n-j} + \tau \beta f_{n+1},$$

где *коэффициенты* выбираются из условий аппроксимации метода. В отличие от *методов типа Рунге - Кутты*, при использовании ФДН - методов нелинейная система алгебраических уравнений для определения u_{n+1} имеет меньшую *размерность*, следовательно, требуется меньшее число операций для нахождения решения.

Условно работу нашей программы можно представить с помощью следующей блок-схемы:



Первый зарегистрированный пользователь- это по умолчанию – администратор. Его данные – логин, пароль и имя почтового ящика сохраняются в базе данных в таблице users. В функции администратора входит создание и загрузка новых уравнений, а точнее программы на языке R решающей соответствующее уравнение из папки где хранятся все наши созданные программы на R. Эти уравнения сохраняются в базе данных в таблице equations. Далее, администратор может решать эту систему, создавая свои наборы параметров a, b, c, которые также сохраняются в базе данных в таблице params. При желании наборы параметров можно изменять или удалять, они также будут изменяться или удаляться в базе данных.

Решая систему уравнений, наша программа на R создает на выходе два файла с результатами: таблица с данными X, Y, Z от t и файл расширения (.png) с четырьмя графиками зависимостей X(t), Y(t), Z(t), Z(X). Эти результаты и видит пользователь на веб-сайте после нажатия кнопки “Решить уравнение”.

LARAVEL, PHP

Миграции БД

Мы используем миграцию для определения таблицы базы данных для хранения всех наших задач. Миграции БД в Laravel позволяют простым способом определить структуру таблицы базы данных и выполнять модификации с использованием простого и выразительного PHP кода. Вместо того чтобы вручную добавлять столбцы в свои локальные копии БД, вы можете просто запустить миграции, которые мы поместили в систему управления версиями.

Таблица `users`

Пользователь должен иметь возможность создать свой аккаунт в приложении, поэтому нам нужна таблица для хранения данных пользователей. Laravel уже поставляется с миграцией, включающей в себя базовую таблицу `users`. Поэтому нам не нужно вручную её создавать. По умолчанию миграция для таблицы `users` находится в каталоге `database/migrations`.

Таблицы `equations` и `params`

Теперь создадим таблицу, которая будет содержать все наши задачи. Для создания различных классов может быть использован интерфейс Artisan. Он избавит нас от ручной генерации кода при создании проектов Laravel. Используем команду `make:migration` для создания миграции новой базы данных для нашей таблицы `equations`:

```
php artisan make:migration create_equations_table --create=tasks
```

Миграция будет помещена в каталог `database/migrations` нашего проекта. Команда `make:migration` уже добавила автоинкрементный ID и метки времени к файлу миграции. Отредактируем этот файл и добавим дополнительный столбец `name` для имён наших задач, description для

описания и file-location для описания папки где буду храниться наша программа на R.

Таблица users:

Столбец e-mail будет проверяться на уникальность, чтобы не было зарегистрированных пользователей с одинаковой почтой.

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('users');
    }
}
```

Таблица equations:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class MigrateEquationsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('equations', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->text('description');
            $table->string('file_location');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('equations');
    }
}
```

Таблица params:

Мы создаем три столбца с параметрами a,b,c: p1, p2,p3;

Чтобы проверить их на уникальность мы создаем один массив со всеми параметрами одновременно в таблице text. С помощью таблиц user_id и equation_id мы связываем params с users и equations соответственно.

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class MigrateParamsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('params', function (Blueprint $table) {
            $table->increments('id');
            $table->double('p1');
            $table->double('p2');
            $table->double('p3');
            $table->text('unique');
            $table->integer('user_id');
            $table->integer('equation_id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('params');
    }
}
```

Чтобы запустить нашу миграцию, мы будем использовать команду Artisan **`migrate`**. Если вы используете Homestead, вы должны выполнить

эту команду на своей виртуальной машине, так как у вашей host-машины не будет прямого доступа к базе данных:

```
php artisan migrate
```

Эта команда создаст все наши таблицы БД.

Модели Eloquent

Eloquent — это стандартное ORM для Laravel (объектно-реляционное отображение). Eloquent делает безболезненным получение и хранение данных в нашей базе данных, используя чётко определённые «модели». Обычно, каждая Eloquent модель однозначно соответствует одной таблице базы данных.

Модель `User`

В первую очередь нам нужна модель, соответствующая нашей таблице `users`. Однако, если вы зайдете в папку app вашего проекта, вы увидите, что Laravel уже поставляется в комплекте с моделью user, поэтому нам не нужно создавать её вручную.

Модель `equations/params`

Мы снова можем использовать команду Artisan, чтобы сгенерировать эту модель. В этом случае мы будем использовать команду `make:model`:

```
php artisan make:model Equations
```

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Equations extends Model
{
  /**
   * Массово присваиваемые атрибуты.
   *
   * @var array
   */
  protected $fillable = ['name'];
}
```

Отношения Eloquent

Теперь, когда наши модели определены, нам нужно связать их. Например, наш `User` может иметь несколько `Params`, в то время как `Params` привязан к одному `User`. Определение взаимосвязи позволит нам быстро проходить через наши отношения:

```
$user = App\User::find(1);

foreach ($user->params as $task) {
  echo $task->name;
}
```

Отношение `equations/params`

Во-первых, давайте определим отношение для нашей модели `User`. Отношения Eloquent определены как методы моделей. Eloquent поддерживает несколько различных типов отношений. Мы определим функцию `equations` в модели `User`, которая вызывает Eloquent-метод `hasMany()`:

```
<?php

namespace App;

// Импортирования пространства имён...

class User extends Model implements AuthenticatableContract,
                                    AuthorizableContract,
                                    CanResetPasswordContract
{
  use Authenticatable, Authorizable, CanResetPassword;

  // Другие Eloquent свойства...

  /**
   * Получить все задачи пользователя.
   */
  public function equations()
  {
    return $this->hasMany(Task::class);
  }
}
```

Отношение `user`

Теперь давайте определим отношение `user` для модели `params`. И снова мы определим отношение как метод модели. В этом случае мы будем использовать Eloquent-метод `belongsTo()`, определяющий отношение:

```php
<?php

namespace App;

use App\User;
use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
  /**
   * Массово присваиваемые атрибуты.
   *
   * @var array
   */
  protected $fillable = ['name'];

  /**
   * Получить пользователя - владельца данной задачи
   */
  public function user()
  {
    return $this->belongsTo(User::class);
  }
}
```

Маршрутизация

В данном приложении мы будем использовать *контроллеры* для организации наших маршрутов. Контроллеры позволят нам лучше организовать логику HTTP-запросов в нескольких файлах.

У нас будет один маршрут, использующий замыкание: наш маршрут `/`, представляющий из себя лендинг для гостей приложения. Заполним наш маршрут `/`. По этому маршруту мы хотим отрисовывать HTML-шаблон, который содержит страницу «welcome».

В Laravel все HTML-шаблоны хранятся в каталоге `resources/views`, и мы можем использовать вспомогательную функцию `view()`, чтобы возвратить один из этих шаблонов по нашему маршруту:

```php
Route::get('/', function () {
  return view('welcome');
});
```

Аутентификация

Мы должны позволить пользователям создавать учётные записи и входить в наше приложение. Как правило, построение всего слоя аутентификации в веб-приложении является трудоёмкой задачей . Однако, так как это распространённая задача, Laravel делает эту процедуру абсолютно безболезненной.

Во-первых, `app/Http/Controllers/Auth/AuthController` уже включён в приложение Laravel. Этот контроллер использует специальный типаж `AuthenticatesAndRegistersUsers` со всей необходимой логикой для создания и аутентификации пользователей.

Маршруты аутентификации

Нам нужно создать шаблоны регистрации и входа в систему, а также определить маршруты, указывающие на контроллер аутентификации. Для начала добавим нужные нам маршруты в файл `app/Http/routes.php`:

```
// Маршруты аутентификации...
Route::get('auth/login', 'Auth\AuthController@getLogin');
Route::post('auth/login', 'Auth\AuthController@postLogin');
Route::get('auth/logout', 'Auth\AuthController@getLogout');

// Маршруты регистрации...
Route::get('auth/register', 'Auth\AuthController@getRegister');
Route::post('auth/register', 'Auth\AuthController@postRegister');
```

Представления аутентификации

Для аутентификации необходимо создать `login.blade.php` и `register.blade.php` в папке `resources/views/auth`. Конечно, дизайн и стиль этих представлений не имеет значения. Тем не менее, они должны содержать по крайней мере несколько основных полей.

Файл `register.blade.php` должен содержать форму, включающую в себя поля `name`, `email`, `password` `password_confirmation`, эта форма должна создавать `POST` запрос к маршруту `/auth/register`.

Файл `login.blade.php` должен содержать форму, включающую в себя поля `email` и `password`, эта форма должна создавать `POST` запрос к маршруту `/auth/login`.

Контроллер задач

Нам нужно получать и сохранять задачи, поэтому создадим `TaskController` с помощью командной строки Artisan, при этом новый контроллер будет помещён в папку `app/Http/Controllers`:

```
php artisan make:controller TaskController --plain
```

Теперь, когда контроллер создан, создадим стабы для некоторых маршрутов в нашем файле `app/Http/routes.php`, указывающих на контроллер:

```
Route::get('/tasks', 'TaskController@index');
Route::post('/task', 'TaskController@store');
Route::delete('/task/{task}', 'TaskController@destroy');
```

Аутентификация всех маршрутов задач

В данном приложении мы хотим, чтобы все наши маршруты задач требовали аутентификации пользователя. Другими словами, пользователь должен зайти в систему, чтобы создать задачу. Поэтому мы должны ограничить доступ к нашим маршрутам задач и открывать доступ только аутентифицированным пользователям. Laravel контролирует это, используя посредника.

Чтобы проверять аутентификацию пользователя в каждом действии, мы можем добавить вызов метода *`middleware`* в конструктор контроллера. Все доступные посредники маршрута определены в файле `app/Http/Kernel.php`. В данном случае мы хотим добавить посредник `auth` ко всем методам контроллера:

```php
<?php

namespace App\Http\Controllers;

use App\Equation;
use App\Params;
use Illuminate\Http\Request;
use App\Http\Requests;
use Illuminate\Support\Facades\Auth;

class EquationController extends Controller
{

    public function __construct()
    {
        $this->middleware('auth');
    }

    public function selectEquation()
    {
        $equations = Equation::all();
        return view('sel_1', compact('equations'));
    }

    public function addEquation()
    {
        if(Auth::user()->id != 1)
        {
            abort(404);
        }
        return view('addequation');
    }
```

```
    public function storeEquation(Request $request)
    {
        if(Auth::user()->id != 1)
        {
            abort(404);
        }
        $this->validate($request, [
            'name'=>'required',
            'file'=>'required'
        ]);
        move_uploaded_file($_FILES["file"]["tmp_name"], 'R/'.basename($_FILES["file"]["name"]));
        $requestArray = $request->toArray();
        $requestArray['file_location'] = basename($_FILES["file"]["name"]);
        $id = Equation::create($requestArray);
        return redirect()->action('EquationController@selectParams', $id);
    }

    public function selectParams($equation_id)
    {
        $_USERID = Auth::user()->id;
        Equation::findOrFail($equation_id);
        $params = \App\Params::where('equation_id','=',$equation_id)->where('user_id','=',$_USERID)->get();
        return view('sel_2', compact('params','equation_id'));
    }

    public function solve($equation_id, $params_id)
    {
        $USER_ID = Auth::user()->id;
        $equation = Equation::findOrFail($equation_id);
        $param = Params::findOrFail($params_id);
        if($USER_ID != $param->user_id)
        {
            abort(404);
        }



        $retv = "";
        system('Rscript R/'.$equation->file_location.' '.$param->p1.' '.$param->p2.' '.$param->p3,$retv);
        $STR = file_get_contents('table.txt');
        $STR = str_replace(' ', "</td><td>", $STR);
        $STR = str_replace("\n", "</td></tr><tr><td>", $STR);
        return view('result', compact('STR'));
    }


}
```

Создание макетов и представлений

Определяем макет

Почти все веб-приложения используют один макет на всех своих страницах. Например, у нашего приложения есть верхняя панель навигации, которая присутствовала бы на каждой странице (если бы у нас их было больше одной). Laravel упрощает использование этих общих функций на всех страницах, используя **макеты** Blade.

Как мы выяснили ранее, все представления Laravel хранятся в `resources/views`. Давайте определим представление нового макета в `resources/views/layouts/app.blade.php`. Расширение `.blade.php` даёт

фреймворку команду использовать *механизм шаблонной обработки Blade,* чтобы отрисовать это представление.

```
// resources/views/layouts/app.blade.php

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Laravel Quickstart - Intermediate</title>

    <!-- CSS и JavaScript -->
  </head>

  <body>
    <div class="container">
      <nav class="navbar navbar-default">
        <!-- Содержимое Navbar -->
      </nav>
    </div>

    @yield('content')
  </body>
</html>
```

Строчка `@yield('content')` в макете. Это специальная Blade-директива для указания всем дочерним страницам, наследующим этот шаблон, где они могут внедрить своё содержимое.

Функция validate, проверка ввода

Теперь, когда у нас есть форма на нашем представлении, мы должны добавить код к нашему методу`TaskController@store`, чтобы проверить входящие данные из формы и создать новую задачу. Во-первых, давайте проверим ввод.

Для этой формы мы создадим обязательное поле `name` и зададим, что оно должно содержать не более `255`символов. Если проверка не пройдёт, то мы перенаправим пользователя назад к URL `/tasks`, а также возвратим ему в сессию его введённые данные с указанием на ошибки:

```
/**
 * Создание новой задачи.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
```

```
  $this->validate($request, [
    'name' => 'required|max:255',
  ]);

  // Создание задачи...
}
```

Если валидация не пройдена для заданных правил, пользователь будет автоматически перенаправлен туда, откуда он пришёл, и ошибки будут автоматически высвечены в сессии.

Переменная `$errors`

Мы использовали директиву `@include('common.errors')` в нашем представлении, чтобы отобразить ошибки ввода формы. `common.errors` позволяет нам легко показывать ошибки ввода в одинаковом формате на всех наших страницах. Давайте определим содержимое этого представления:

```
// resources/views/common/errors.blade.php

@if (count($errors) > 0)
  <!-- Список ошибок формы -->
  <div class="alert alert-danger">
    <strong>Упс! Что-то пошло не так!</strong>

    <br><br>

    <ul>
      @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif
```

Отображение задач

Когда данные переданы, мы можем обращаться к задачам в нашем представлении `tasks/index.blade.php` и выводить их на экран таблицей. Blade-конструкция `@foreach` позволяет нам кратко писать циклы, которые компилируются в молниеносный простой PHP-код:

```
@extends('layouts.app')

@section('content')
  <!-- Форма создания задачи... -->

  <!-- Текущие задачи -->
  @if (count($tasks) > 0)
    <div class="panel panel-default">
      <div class="panel-heading">
        Текущая задача
      </div>

      <div class="panel-body">
        <table class="table table-striped task-table">

          <!-- Заголовок таблицы -->
          <thead>
            <th>Task</th>
            <th> </th>
          </thead>

          <!-- Тело таблицы -->
          <tbody>
            @foreach ($tasks as $task)
              <tr>
                <!-- Имя задачи -->
                <td class="table-text">
                  <div>{{ $task->name }}</div>
                </td>

                <td>
                  <!-- TODO: Кнопка Удалить -->
                </td>
              </tr>
            @endforeach
          </tbody>
        </table>
      </div>
    </div>
   @endif
@endsection
```

Удаление задач

Добавление кнопки удаления задачи

Теперь добавим кнопку удаления к каждой строке нашего списка задач в представлении `tasks/index.blade.php`. Мы создадим маленькую однокнопочную форму для каждой задачи в списке. После нажатия кнопки приложению будет отправляться запрос `DELETE /task`, который будет обращаться к методу `TaskController@destroy`:

```
<tr>
  <!-- Имя задачи -->
   <td class="table-text">
    <div>{{ $task->name }}</div>
  </td>

  <!-- Кнопка Удалить -->
  <td>
    <form action="/task/{{ $task->id }}" method="POST">
      {{ csrf_field() }}
      {{ method_field('DELETE') }}

      <button>Удалить задачу</button>
    </form>
  </td>
</tr>
```

Примечание по спуфингу метода

Обратите внимание на то, что `method` формы кнопки удаления объявлен как `POST`, несмотря на то, что мы отвечаем на запрос, используя маршрут `Route::delete`. HTML-формы позволяют использовать только `GET` и `POST` методы HTTP. А нам нужен способ имитировать запрос `DELETE` от формы.

Мы можем имитировать запрос `DELETE`, выводя результаты функции `method_field('DELETE')` в нашей форме. Эта функция генерирует скрытый ввод формы, который распознается Laravel и используется, чтобы переопределить вызываемый метод HTTP. Сгенерированное поле будет похоже на это:

```
<input type="hidden" name="_method" value="DELETE">
```

Заключение

Был создан удобный и простой в использовании web-интерфейс для решения системы дифференциальных уравнений, который при дальнейшем развитии может расширять диапазон своих функций и оказывать помощь в решении уравнений и исследовании физических явлений.

Полученные данные (значения и графики) были проверены в программе Mathematica и являются надежными.

У данной программы также есть множество перспектив, например:

- Возможность выбора разных видов дифференциальных уравнений
- Увеличение количества начальных параметров, задаваемых пользователем
- Возможность выбора зависимости для построения графиков
- Возможность сохранять полученные таблицы и графики пользователем в файл.

Список использованной литературы:

- Р.И.Кабаков. R в действии. Анализ и визуализация данных на языке R. Москва,2014.
- А.Б.Шипунов, Е.М.Балдин, П.А.Волкова, А.И.Коробейников, С.А.Назарова, С.В. Петров, А.Г.Суфиянов. Наглядна статистика. Используем R! –М.: ДМК Пресс, 2012. – 298 с.: ил. – ISBN 978-5-94074-785-828-1
- https://cran.r-project.org/web/packages/deSolve/vignettes/deSolve.pdf
- http://w.ict.nsc.ru/books/textbooks/akhmerov/ode/s-17/s-17.html
- http://www.math.rsu.ru/mexmat/kvm/MME/dsarch/Lorenz.html
- https://laravel.com/
- http://laravel.su/