RS-Express Bus Booking System - Development Guide

Table of Contents

Development Environment Setup

Prerequisites

- PHP 8.1 or higher
- Composer
- Node.js 16 or higher
- npm or Yarn
- MySQL 5.7 or higher
- Git

Backend Setup

1. Clone the repository:

```
git clone https://github.com/your-repo/rs-express.git
cd rs-express/Back-end(working)
```

2. Install PHP dependencies:

```
composer install
```

3. Create environment file:

```
cp .env.example .env
```

4. Configure database connection in `.env`:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=rs_express
DB_USERNAME=your_username
DB_PASSWORD=your_password
```

5. Generate application key:

```
php artisan key:generate
```

6. Run migrations and seed the database:

```
php artisan migrate --seed
```

7. Start Laravel server:

```
php artisan serve
```

Frontend Setup

1. Navigate to frontend directory:

```
cd ../bus(working)
```

2. Install JavaScript dependencies:

```
npm install
```

3. Create `.env.local` file:

```
REACT_APP_API_URL=http://localhost:8000/api
```

4. Start development server:

```
npm start
```

Project Structure

Backend (Laravel) Structure

The backend follows a standard Laravel structure with some specific organization:

```
Back-end(working)/
├── app/
│   ├── Console/            # Command line tasks
│   ├── Http/
│   │   ├── Controllers/    # Request handlers
│   │   ├── Middleware/     # Request filters
│   │   └── Requests/       # Form validation
│   ├── Models/             # Eloquent models
│   └── Services/           # Business logic services
├── config/                 # Configuration files
├── database/
│   ├── factories/          # Model factories for testing
│   ├── migrations/         # Database schema definitions
│   └── seeders/            # Database seeders
├── routes/                 # API and web route definitions
│   ├── api.php             # API routes
│   ├── web.php             # Web routes
│   └── auth.php            # Authentication routes
└── tests/                  # Automated tests
```

Frontend (React) Structure

The frontend follows a feature-based organization:

```
bus(working)/
├── public/                 # Static assets
├── src/
│   ├── components/         # Reusable UI components
│   ├── context/            # React Context providers
│   ├── pages/              # Page components
│   ├── services/           # API service modules
│   ├── utils/              # Utility functions
│   ├── App.js              # Main application component
│   └── index.js            # Application entry point
└── package.json            # Project dependencies
```

Coding Standards

PHP Coding Standards

- Follow PSR-12 coding standard
- Use meaningful variable and function names
- Add DocBlocks to methods and classes
- Keep methods small and focused on a single responsibility
- Use type hinting where possible
- Use Laravel conventions for naming (e.g., controllers, models)

Example controller method:

```
/**
 * Store a new booking in the database.
 *
 * @param  \App\Http\Requests\StoreBookingRequest  $request
 * @return \Illuminate\Http\JsonResponse
 */
public function store(StoreBookingRequest $request)
{
    $booking = $this->bookingService->createBooking($request->validated());

    return response()->json([
        'status' => 'success',
        'message' => 'Booking created successfully',
        'data' => $booking
    ], 201);
}
```

JavaScript Coding Standards

- Use ES6+ features
- Follow airbnb style guide
- Use functional components with hooks
- Add JSDoc comments for functions
- Use camelCase for variables and functions
- Use PascalCase for components
- Keep components focused on a single responsibility

Example React component:

```
/**
 * BookingForm component for creating new bookings
 * @param {Object} props - Component props
 * @param {function} props.onSubmit - Form submission handler
 * @param {Object} props.initialValues - Initial form values
 */
const BookingForm = ({ onSubmit, initialValues = {} }) => {
  const [formData, setFormData] = useState(initialValues);

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData(prev => ({ ...prev, [name]: value }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    onSubmit(formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* Form fields */}
    </form>
  );
};

export default BookingForm;
```

Backend Development

Creating Models

1. Generate a model with migration:

```
php artisan make:model ModelName -m
```

2. Define fields in migration:

```
Schema::create('model_names', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->text('description')->nullable();
    $table->foreignId('user_id')->constrained();
    $table->timestamps();
});
```

3. Define relationships in model:

```
public function user()
{
    return $this->belongsTo(User::class);
}

public function items()
{
    return $this->hasMany(Item::class);
}
```

Creating Controllers

1. Generate a resource controller:

```
php artisan make:controller ControllerName --resource --model=ModelName
```

2. Implement required methods:

```
public function index()
{
    return ModelName::paginate(15);
}

public function store(StoreModelNameRequest $request)
{
    $model = ModelName::create($request->validated());
    return response()->json($model, 201);
}
```

Form Request Validation

1. Generate a form request:

```
php artisan make:request StoreModelNameRequest
```

2. Define validation rules:

```
public function rules()
{
    return [
        'name' => 'required|string|max:255',
        'description' => 'nullable|string',
        'user_id' => 'required|exists:users,id',
    ];
}
```

Creating Services

1. Create a service class:

```
namespace App\Services;

class BookingService
{
    /**
     * Create a new booking
     */
    public function createBooking(array $data)
    {
        // Business logic
        $booking = Booking::create($data);

        // Additional processing
        event(new BookingCreated($booking));

        return $booking;
    }
}
```

2. Use dependency injection in controllers:

```
public function __construct(private BookingService $bookingService)
{
    $this->bookingService = $bookingService;
}
```

Frontend Development

Creating Components

1. Create a component file:

```
// src/components/BookingCard.js
import React from 'react';
import PropTypes from 'prop-types';

const BookingCard = ({ booking, onCancel }) => {
  return (
    <div className="booking-card">
      <h3>{booking.route_name}</h3>
      <p>Date: {booking.departure_date}</p>
      <p>Seat: {booking.seat_no}</p>
      <button onClick={() => onCancel(booking.id)}>
        Cancel Booking
      </button>
    </div>
  );
};

BookingCard.propTypes = {
  booking: PropTypes.object.isRequired,
  onCancel: PropTypes.func.isRequired,
};

export default BookingCard;
```

Managing State with Context

1. Create a context file:

```
// src/context/AuthContext.js
import React, { createContext, useState, useEffect } from 'react';
import authService from '../services/authService';

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const token = localStorage.getItem('token');
    if (token) {
      authService.getCurrentUser()
        .then(userData => {
          setUser(userData);
          setLoading(false);
        })
        .catch(() => {
          localStorage.removeItem('token');
          setLoading(false);
        });
    } else {
      setLoading(false);
    }
  }, []);

  const login = async (credentials) => {
    const response = await authService.login(credentials);
    localStorage.setItem('token', response.token);
    setUser(response.user);
    return response.user;
  };

  const logout = () => {
    authService.logout();
    localStorage.removeItem('token');
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, loading, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};
```

2. Use the context in components:

```
import React, { useContext } from 'react';
import { AuthContext } from '../context/AuthContext';

const ProfilePage = () => {
  const { user, logout } = useContext(AuthContext);

  if (!user) {
    return <p>Please log in</p>;
  }

  return (
    <div>
      <h1>Welcome, {user.name}</h1>
      <button onClick={logout}>Logout</button>
    </div>
  );
};
```

API Services

1. Create service modules:

```
// src/services/bookingService.js
import axios from '../utils/axiosConfig';

const bookingService = {
  getAllBookings: async () => {
    const response = await axios.get('/bookings');
    return response.data;
  },

  createBooking: async (bookingData) => {
    const response = await axios.post('/bookings', bookingData);
    return response.data;
  },

  cancelBooking: async (id, reason) => {
    const response = await axios.post(`/bookings/${id}/cancel`, { reason });
    return response.data;
  }
};

export default bookingService;
```

2. Configure Axios with interceptors:

```
// src/utils/axiosConfig.js
import axios from 'axios';

const instance = axios.create({
  baseURL: process.env.REACT_APP_API_URL,
  headers: {
    'Content-Type': 'application/json'
  }
});

// Add token to requests
instance.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers['Authorization'] = `Bearer ${token}`;
  }
  return config;
});

// Handle response errors
instance.interceptors.response.use(
  response => response,
  error => {
    if (error.response && error.response.status === 401) {
      localStorage.removeItem('token');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);

export default instance;
```

Testing

Backend Testing

1. Set up testing environment:

```
cp .env .env.testing
```

2. Configure `.env.testing`:

```
DB_CONNECTION=sqlite
DB_DATABASE=:memory:
```

3. Write feature tests:

```
namespace Tests\Feature;

use Tests\TestCase;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;

class BookingTest extends TestCase
{
    use RefreshDatabase;

    public function test_user_can_create_booking()
    {
        $user = User::factory()->create();

        $response = $this->actingAs($user)
            ->postJson('/api/bookings', [
                'trip_id' => 1,
                'seat_no' => 'A1'
            ]);

        $response->assertStatus(201)
            ->assertJsonStructure([
                'status',
                'message',
                'data' => [
                    'id',
                    'seat_no',
                    'status'
                ]
            ]);
    }
}
```

4. Run tests:

```
php artisan test
```

Frontend Testing

1. Write component tests with React Testing Library:

```
// src/components/__tests__/BookingForm.test.js
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import BookingForm from '../BookingForm';

describe('BookingForm', () => {
  test('submits the form with correct values', () => {
    const handleSubmit = jest.fn();

    render(<BookingForm onSubmit={handleSubmit} />);

    fireEvent.change(screen.getByLabelText(/seat number/i), {
      target: { value: 'A1' }
    });

    fireEvent.submit(screen.getByRole('button', { name: /book now/i }));

    expect(handleSubmit).toHaveBeenCalledWith(
      expect.objectContaining({ seat_no: 'A1' })
    );
  });
});
```

2. Run tests:

```
npm test
```

Deployment

Backend Deployment

1. Set up production server with PHP, Nginx/Apache, MySQL
2. Configure server environment variables
3. Deploy code:

```
git pull origin main
composer install --optimize-autoloader --no-dev
php artisan config:cache
php artisan route:cache
php artisan view:cache
php artisan migrate --force
```

Frontend Deployment

1. Build production assets:

```
npm run build
```

2. Deploy `build` directory to static hosting or CDN

Version Control

Branching Strategy

- `main`: Production-ready code
- `develop`: Integration branch for features
- `feature/*`: New features
- `bugfix/*`: Bug fixes
- `hotfix/*`: Critical production fixes

Commit Conventions

Follow conventional commits:

- `feat:` New features
- `fix:` Bug fixes
- `docs:` Documentation changes
- `style:` Code style changes
- `refactor:` Code refactoring
- `test:` Test updates
- `chore:` Build process or tooling changes

Example:

```
feat(booking): add cancellation feature

- Add cancellation endpoint
- Create cancellation form component
- Implement refund calculation
```

Contribution Guidelines

Workflow

1. Fork the repository
2. Create a feature branch
3. Develop and test your changes
4. Submit a pull request to the `develop` branch
5. Code review
6. Merge into `develop`

Pull Request Requirements

- Pass all tests
- Follow coding standards
- Include appropriate tests
- Update documentation as needed
- Complete pull request template

Code Review Checklist

- Functional requirements met
- Code quality and style
- Test coverage
- Security considerations
- Performance impact
- Documentation updates