

PROJECT TITLE: CONCISEA (BITLY CLONE)

*A Thesis of Experiential Learning
Submitted in partial fulfilment of the requirements for the degree of B.Tech. in
CSE*

| NAME | CRANES REGD. NO. | Regd. No. |
|----------------------|---------------------|------------|
| CHANDAN CHOUDHURY | CL2025010601911046 | 2201020352 |
| ADITYA NAYAK | CL2025010601911349 | 2201020377 |
| SANJEEB KUMAR POTHAL | CL2025010601938185 | 2201020411 |
| GYANA RANJAN SAHOO | CL20250106019414118 | 2201020033 |

Dept. of CSE C.V. Raman Global University, Odisha



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
C. V. Raman Global University, Odisha, Bhubaneswar
PIN -752054, India

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to all those who contributed to the successful completion of our project titled "**Concisea (Bitly Clone)**" Using Java. This Project is submitted in partial fulfilment of the requirements for the degree of **B.Tech. in Computer Science and Engineering**.

We are also grateful to the faculty members of the **Department of Computer Science and Engineering** at **C.V. Raman Global University**, Bhubaneswar, for providing us with the knowledge and resources required for this project. Their insights and suggestions have greatly enriched our work.

Furthermore, we would like to express our appreciation to our friends and classmates for their constant support and motivation during challenging times. Special thanks to our families for their unwavering love, patience, and encouragement throughout this journey.

Lastly, we acknowledge the contribution of various online resources, documentation, and open-source libraries that aided in the successful implementation of the project.

Team Members:

- **Chandan Choudhury** (CRANES Regd. No. CL2025010601911046), 2201020352
- **Sanjeeb Kumar Pothal** (CRANES Regd. No. CL2025010601938185), 2201020411
- **Aditya Nayak** (CRANES Regd. No. CL2025010601911349), 2201020377
- **Gyana Ranjan Sahoo** (CRANES Regd. No. CL20250106019414118), 2201020033

We sincerely hope that our project will prove beneficial for future studies and inspire others to explore similar technological solutions.

Thank you.

TABLE OF CONTENTS

| SL NO. | CONTENT | PAGE NO. |
|---------------|--|-----------------|
| 1 | Introduction and Project Setup | 5 |
| 2 | Database Design and Configuration | 6 - 7 |
| 3 | Authentication and Security | 8 - 9 |
| 4 | User Management and Access Control | 10 - 11 |
| 5 | URL Shortening Logic Implementation | 12 |
| 6 | Analytics and Tracking System | 13 - 14 |
| 7 | Frontend Development with React and Tailwind CSS | 15 - 17 |
| 8 | Testing and Error Handling | 18 - 19 |
| 9 | Outputs and Results | 20 - 22 |
| 10 | Future Integrations | 23 |
| 11 | Conclusion and References | 24 |

ABSTRACT

The project titled "**Concisea (Bitly Clone)**" Using Java presents a comprehensive solution for URL shortening, focusing on user convenience, security, and efficient management. In the digital landscape, lengthy URLs often pose challenges in sharing, tracking, and memorizing web addresses. This project addresses those issues by developing a robust web-based platform that enables users to generate concise URLs while providing features like click tracking, analytics, and user authentication.

The system generates unique short URLs and maintains a database for tracking the original URLs. Users can create and manage shortened links, while basic analytics track the number of times a link has been accessed. The project currently lacks features such as website hosting, a "Forgot Password" option, custom short URLs, and advanced analytics.

Future enhancements could include **detailed analytics with location-based tracking, browser and device insights, an authentication recovery system, and online deployment**. This project serves as a foundation for further development, showcasing the capabilities of modern web technologies in URL management.

CHAPTER 1: INTRODUCTION AND PROJECT SETUP

1.1 Introduction to URL Shortening URL shortening services transform lengthy URLs into concise, manageable links. These services are widely used in social media, marketing campaigns, and email communications to improve link accessibility and tracking capabilities. By compressing URLs into shorter versions, users can enhance user engagement while preserving the original destination.

1.2 Importance of URL Shorteners URL shorteners are essential for improving link management by providing clickable, trackable, and shareable links. They simplify long web addresses, enhance readability, and enable detailed analytics to monitor user engagement and link performance.

1.3 Objectives of the Project

- Develop a secure and scalable URL shortening application using Java and Spring Boot.
- Implement JWT-based authentication to ensure secure user access.
- Enable URL analytics to provide users with insights into their link performance.
- Design a responsive user interface using React and Tailwind CSS.
- Scalability and cloud integrated Database.

1.4 Project Scope The project covers:

- Creating and managing shortened URLs.
- Implementing user authentication for security.
- Providing analytics dashboards for user insights.
- Ensuring smooth deployment with cloud integration.

1.5 Tools and Technologies Used

- **Frontend:** React, Tailwind CSS
- **Backend:** Java, Spring Boot, JWT Authentication
- **Database:** PostgreSQL
- **Deployment Tools:** Cloud Database Integration
- **Developed on:** IntelliJ (Backend), VS Code (Frontend)

CHAPTER 2: DATABASE DESIGN AND CONFIGURATION

2.1 Database Schema Design

A well-structured database schema is crucial for efficient data management, seamless system performance, and ensuring data integrity. The schema for the URL shortening application is designed to optimize storage, retrieval, and security of user data and URL information. The key tables in the schema include:

- **Users Table:** Stores user credentials, hashed passwords, and role information.
- **URL Mapping Table:** Maintains records of shortened URLs, their original links, creation timestamps, and user associations.
- **Click Event Table:** Captures user activity such as click count and timestamps for comprehensive insights.

Each table is linked using appropriate relationships to maintain referential integrity and improve query performance. The schema follows a one-to-many relationship structure, ensuring efficient data organization.

2.2 Tables for URL Management

Effective URL management is vital for tracking and controlling shortened links. The key attributes of the URL Mapping table include:

- **ID:** Unique identifier for each URL entry.
- **Original URL:** The long-form web address provided by the user.
- **Shortened URL:** The generated unique short link.
- **Created Date:** Tracks when the URL was created.
- **User ID:** Associates the URL with a specific user.
- **Click Count:** Tracks the number of times the shortened URL has been accessed.

The Click Event table stores:

- **Click ID:** Unique identifier for each click event.
- **Click Date:** Timestamp of when the URL was accessed.
- **URL Mapping ID:** References the respective URL entry to track its interactions.

2.3 Establishing Database Connections

Establishing secure and efficient database connections is essential for seamless communication between the backend logic and the PostgreSQL database. The steps involved include:

- **Step 1:** Configure database dependencies in `pom.xml` (for Maven).
- **Step 2:** Define database configurations in `application.properties` specifying essential details such as:

```
spring.datasource.url=${DATABASE_URL}

spring.datasource.username=${DATABASE_USERNAME}

spring.datasource.password=${DATABASE_PASSWORD}

spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=${DATABASE_DIALECT}
```

- **Step 3:** Implement `@Entity` annotations in Java classes to map the database tables directly with the application's object model.
- **Step 4:** Use `@Repository` to create Data Access Objects (DAO) for streamlined database interactions.

2.4 Environment Configuration

The `env.prod` file is used to securely store database and authentication configurations:

```
DATABASE_URL=jdbc:postgresql://<HOST>/<DATABASE_NAME>?user=<USERNAME>&password=<PASSWORD>&sslmode=<SSLMODE>DATABASE_USERNAME=urlshortenerdb_owner
```

```
DATABASE_PASSWORD=npg_7SyitP0AjbBD
```

```
DATABASE_DIALECT=org.hibernate.dialect.PostgreSQLDialect
```

```
JWT_SECRET=0f60440446806a2eba0ba9ab9c0d0ee3f0d8957f1791014d673a633c1577576f30b6b8f66e7ae8c6cbc9a97ee87ed8a8b6ed91b1982128445f26d2820ef4a071
```

```
FRONTEND_URL=http://localhost:5173
```

This configuration ensures secure database communication, proper ORM handling, and JWT-based authentication for user sessions. Debugging and logging settings are also configured to monitor application performance effectively.

CHAPTER 3: AUTHENTICATION AND SECURITY

3.1 JWT Authentication Overview

JSON Web Token (JWT) is a compact, URL-safe mechanism for securely transmitting authentication data between a client and a server. It is widely used for web application authentication due to its stateless nature, reducing server load and improving performance. Each JWT consists of:

- **Header:** Specifies the token type (JWT) and the signing algorithm used.
- **Payload:** Contains user-related claims such as user ID, role, and token expiration time.
- **Signature:** Ensures the integrity and authenticity of the token using a secret key.

3.2 Implementing JwtUtil for Token Management

The `JwtUtil` class is responsible for handling JWT token operations, including token creation, validation, and extracting claims. Key methods include:

- `generateToken()`: Creates a JWT for a user based on authentication credentials and expiration rules.
- `validateToken()`: Verifies the authenticity and validity of the received token.
- `extractUsername()`: Retrieves the username from the token payload for user identification.

Example Code Snippet:

```
public String generateToken(UserDetails userDetails) {  
    return Jwts.builder()  
        .setSubject(userDetails.getUsername())  
        .setIssuedAt(new Date())  
        .setExpiration(new Date(System.currentTimeMillis() +  
TOKEN_EXPIRATION_TIME))  
        .signWith(SignatureAlgorithm.HS512, SECRET_KEY)  
        .compact();  
}
```

3.3 Creating JwtAuthenticationFilter for Security

The `JwtAuthenticationFilter` class intercepts incoming HTTP requests and verifies JWT tokens. Key functions include:

- Extracting tokens from the Authorization header.
- Validating tokens using `JwtUtil`.
- Setting up authentication in the `SecurityContextHolder`.

Example Code Snippet:

```
@Override
```



```
protected void doFilterInternal(HttpServletRequest request,
                                HttpServletResponse response,
                                FilterChain filterChain)
                                throws ServletException, IOException {
    String token = extractTokenFromHeader(request);
    if (token != null && jwtUtil.validateToken(token)) {
        UsernamePasswordAuthenticationToken authentication =
            new UsernamePasswordAuthenticationToken(
                jwtUtil.extractUsername(token), null, new ArrayList<>());
        SecurityContextHolder.getContext().setAuthentication(authentication);
    }
    filterChain.doFilter(request, response);
}
```

3.4 Securing Endpoints with Spring Security

Spring Security is configured to protect application endpoints by enforcing authentication and authorization. Key configurations include:

- Defining public and private endpoints.
- Integrating `JwtAuthenticationFilter` into the security pipeline.
- Implementing role-based access control.

Example Code Snippet:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilterBefore(jwtAuthenticationFilter(),
                UsernamePasswordAuthenticationFilter.class);
    }
}
```

By implementing JWT authentication, request filtering, and endpoint security, the system ensures secure access management, data protection, and improved performance for the crowdfunding platform.

CHAPTER 4: USER MANAGEMENT AND ACCESS CONTROL

4.1 Implementing UserDetailsServiceImpl

The `UserDetailsServiceImpl` class is responsible for managing user authentication and authorization in the system. It implements Spring Security's `UserDetailsService` interface to load user details from the database during login.

Key Responsibilities:

- **Loading User Data:** Uses `loadUserByUsername()` to fetch user details from the database.
- **User Role Management:** Maps user roles to ensure proper authentication and authorization.
- **Password Validation:** Ensures secure password verification using encryption mechanisms.

Example Code Snippet:

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not
found"));
        return new org.springframework.security.core.userdetails.User(
            user.getUsername(),
            user.getPassword(),
            user.getRoles()
        );
    }
}
```

4.2 Role-based Access Control (RBAC)

Role-based access control (RBAC) ensures that users can only access resources permitted by their assigned roles, enhancing security.

Implementation Steps:

- **Role Definition:** Define roles such as `USER` and `ADMIN` in the database.
- **Role Assignment:** Assign roles during user registration or in the database.
- **Access Control Rules:** Enforce security policies in the `SecurityConfig` class.

Example Code Snippet:

```

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/user/**").hasAnyRole("USER", "ADMIN")
            .anyRequest().authenticated();
    }
}

```

4.3 Managing User Sessions with Tokens

JWT tokens are used to maintain user sessions securely in a stateless architecture.

Key Steps:

- **Token Generation:** Upon login, a JWT token is issued with user details and expiration time.
- **Token Storage:** Tokens are stored securely in browser local storage or cookies.
- **Token Validation:** Each request validates the token using `JwtAuthenticationFilter`.

Example Code Snippet:

```

public String generateToken(UserDetails userDetails) {
    return Jwts.builder()
        .setSubject(userDetails.getUsername())
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() +
TOKEN_EXPIRATION_TIME))
        .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
        .compact();
}

```

By implementing `UserServiceImpl`, securing endpoints using RBAC, and managing user sessions with JWT, the project achieves a secure and structured user management system, ensuring safe authentication and authorization.

CHAPTER 5: URL SHORTENING LOGIC IMPLEMENTATION

5.1 Generating Unique Short URLs The core functionality of the URL shortening application is to generate concise yet unique URLs for efficient redirection. The process involves:

- **Hashing Algorithm:** A hashing technique (e.g., Base62 encoding, MD5, or SHA-256) is used to convert long URLs into shorter unique identifiers.
- **Collision Prevention:** To avoid duplicate shortened URLs, a uniqueness check is implemented before storing entries in the database.
- **URL Format:** The generated short URL follows a structure similar to `https://short.domain/abc123`.

Example Code Snippet for URL Generation:

```
public String generateShortUrl(String originalUrl) {  
    String uniqueId = UUID.randomUUID().toString().substring(0, 8);  
    return BASE_URL + "/" + uniqueId;  
}
```

CHAPTER 6: ANALYTICS AND TRACKING SYSTEM

6.1 Tracking URL Usage

Tracking shortened URL usage helps in monitoring redirection activity and user interactions. The project implements basic tracking mechanisms to record:

- **Short URL Access Frequency:** Logs each time a shortened URL is accessed.
- **Timestamp Logging:** Records the time when the redirection occurs.
- **Unique Visits Count:** Identifies how many unique users accessed the URL.
- **Database Storage:** URL access logs are stored in the database for future reference.

Example Code Snippet for Tracking Access:

```
@GetMapping("/{shortUrl}")
public ResponseEntity<?> redirectToOriginalUrl(@PathVariable String shortUrl)
{
    Optional<UrlEntity> urlEntity = urlService.findByShortUrl(shortUrl);
    if (urlEntity.isPresent()) {
        // Increment access count
        urlEntity.get().incrementAccessCount();
        urlService.save(urlEntity.get());
        return ResponseEntity.status(HttpStatus.FOUND)
            .header("Location", urlEntity.get().getOriginalUrl())
            .build();
    }
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body("URL Not Found");
}
```

6.2 Generating Basic Analytical Reports

The system generates simple analytical reports by collecting URL access statistics. The reports include:

- **Total Clicks:** Displays the total number of times a shortened URL was accessed.
- **Unique Visitors:** Differentiates between repeat visits and new users.
- **Date-based Trends:** Shows URL usage trends over time.

Example Code Snippet for Report Generation:

```
public UrlAnalyticsReport generateReport(String shortUrl) {
    UrlEntity urlEntity = urlService.findByShortUrl(shortUrl).orElseThrow();
    return new UrlAnalyticsReport(urlEntity.getShortUrl(),
        urlEntity.getAccessCount(), urlEntity.getCreatedAt());
}
```

6.3 Visualizing URL Performance

A dashboard can be created to display analytics data in a user-friendly manner. Features include:

- **Bar Charts for Click Trends:** Displays total URL clicks over time.
- **Data Tables for Usage Metrics:** Lists URLs with their access counts.

- **Filters for Custom Reports:** Allows filtering data by date or specific URLs.

Example Code Snippet for Frontend Dashboard:

```
<BarChart data={clickData} options={{ title: 'URL Click Trends' }} />  
<DataTable data={urlAnalytics} columns={['Short URL', 'Total Clicks',  
'Created At']} />
```

By implementing URL tracking, generating reports, and visualizing performance metrics, the project ensures enhanced monitoring of URL usage and improved user insights.

CHAPTER 7: FRONTEND DEVELOPMENT WITH REACT AND TAILWIND CSS

7.1 Building the Landing Page

The landing page is the first interaction point for users, providing an intuitive interface for URL shortening. Key features include:

- **Minimalist UI:** A clean, responsive interface designed with Tailwind CSS.
- **Primary Call-to-Action (CTA):** Users can input their long URL and generate a short URL instantly.
- **Navigation Bar:** Provides access to login, signup, and dashboard pages.

Example Code from Project:

```
import React, { useState } from 'react';
const LandingPage = () => {
  const [longUrl, setLongUrl] = useState('');
  const handleSubmit = async (e) => {
    e.preventDefault();
    // Call backend API to shorten URL
  };
  return (
    <div className="min-h-screen flex flex-col items-center justify-
center bg-gray-100">
      <h1 className="text-3xl font-bold mb-6">Shorten Your URLs</h1>
      <form onSubmit={handleSubmit} className="flex space-x-2">
        <input
          type="text"
          placeholder="Enter URL"
          className="p-2 border rounded w-80"
          value={longUrl}
          onChange={(e) => setLongUrl(e.target.value)}
        />
        <button type="submit" className="bg-blue-500 text-white px-4
py-2 rounded">Shorten</button>
      </form>
    </div>
  );
};
export default LandingPage;
```

7.2 Implementing Signup and Login Pages

User authentication ensures that only registered users can manage their URLs. Features include:

- **Signup and Login Forms:** Secure forms with field validation.
- **API Integration:** Calls backend for authentication.
- **Error Handling:** Displays appropriate error messages.

Example Code from Project:

```
import React, { useState } from 'react';
const Signup = () => {
  const [formData, setFormData] = useState({ username: '', email: '',
password: '' });
  const handleChange = (e) => setFormData({ ...formData, [e.target.name]:
e.target.value });
  const handleSubmit = async (e) => {
    e.preventDefault();
    // API Call to register user
  };
  return (
    <form onSubmit={handleSubmit} className="max-w-md mx-auto p-4 bg-
gray-100 shadow rounded">
      <input name="username" placeholder="Username"
onChange={handleChange} className="block w-full mb-3 p-2 border" />
      <input name="email" type="email" placeholder="Email"
onChange={handleChange} className="block w-full mb-3 p-2 border" />
      <input name="password" type="password" placeholder="Password"
onChange={handleChange} className="block w-full mb-3 p-2 border" />
      <button type="submit" className="bg-blue-500 text-white px-4 py-2
rounded">Register</button>
    </form>
  );
};
export default Signup;
```

7.3 Developing the Dashboard and URL Management

The dashboard allows users to manage their shortened URLs, track visits, and delete links. Features include:

- **URL Management:** Displays active shortened URLs.
- **Click Tracking:** Shows total clicks on each URL.
- **Delete Option:** Users can remove old or unused URLs.

Example Code from Project:

```
import React, { useEffect, useState } from 'react';
const Dashboard = () => {
  const [urls, setUrls] = useState([]);
  useEffect(() => {
    // Fetch URLs from backend
  }, []);
  return (
    <div className="p-4">
      <h2 className="text-2xl font-bold">Your Shortened URLs</h2>
      <ul className="mt-4">
        {urls.map((url) => (
          <li key={url.id} className="mb-2 p-2 border rounded flex
justify-between">
```



```

                                <a href={url.shortUrl} target="_blank" rel="noopener
noreferrer" className="text-blue-600">{url.shortUrl}</a>
                                <span>Clicks: {url.clicks}</span>
                                <button className="bg-red-500 text-white px-2 py-1
rounded">Delete</button>
                                </li>
                                )})
                                </ul>
                                </div>
                                );
};
export default Dashboard;

```

By implementing a structured frontend with **React and Tailwind CSS**, the project ensures a **user-friendly, responsive, and efficient** URL shortening experience.

CHAPTER 8: TESTING AND ERROR HANDLING

8.1 Implementing Error Responses Proper error handling is crucial for ensuring a smooth user experience and system reliability. The following strategies are implemented:

- **Custom Error Messages:** The application provides meaningful error messages to guide users when issues arise.
- **Centralized Exception Handling:** Error handling is implemented in a centralized manner to catch unexpected failures.
- **Status Codes:** HTTP status codes such as 400 Bad Request, 404 Not Found, and 500 Internal Server Error are used appropriately.

Example Code Snippet for Error Handling:

```
app.use((err, req, res, next) => {  
  console.error(err.stack);  
  res.status(500).json({ error: "Something went wrong! Please try again  
later." });  
});
```

8.2 Testing URL Shortening Functionality Testing ensures that the core features of the URL shortener work as expected. The following tests are performed:

- **Short URL Generation Test:** Ensures that valid URLs are shortened correctly and stored in the database.
- **Redirection Test:** Confirms that shortened URLs redirect to their corresponding original URLs.
- **Invalid URL Handling:** Tests how the system handles invalid or expired URLs.

Example Code Snippet for URL Shortening Test:

```
test("Short URL Generation", async () => {  
  const response = await request(app).post("/shorten").send({ url:  
"https://example.com" });  
  expect(response.status).toBe(201);  
  expect(response.body.shortUrl).toBeDefined();  
});
```

8.3 Authentication Testing User authentication is tested to ensure that login and signup functionalities work securely and as expected.

- **Signup Test:** Validates that new users can register successfully.
- **Login Test:** Ensures that users can log in with valid credentials.
- **JWT Authentication:** Confirms that access tokens are correctly generated and validated.

Example Code Snippet for Login Test:

```
test("User Login", async () => {
```

```
const response = await request(app).post("/login").send({ username:
"testuser", password: "password123" });
expect(response.status).toBe(200);
expect(response.body.token).toBeDefined();
});
```

8.4 Ensuring Data Integrity To maintain consistent and accurate data, the following strategies are implemented:

- **Database Constraints:** Primary keys, foreign keys, and unique constraints prevent duplicate or invalid entries.
- **Validation Rules:** Ensures that required fields such as URLs and authentication details follow a standard format.
- **Data Cleanup:** Regular database maintenance ensures the removal of expired and redundant data.

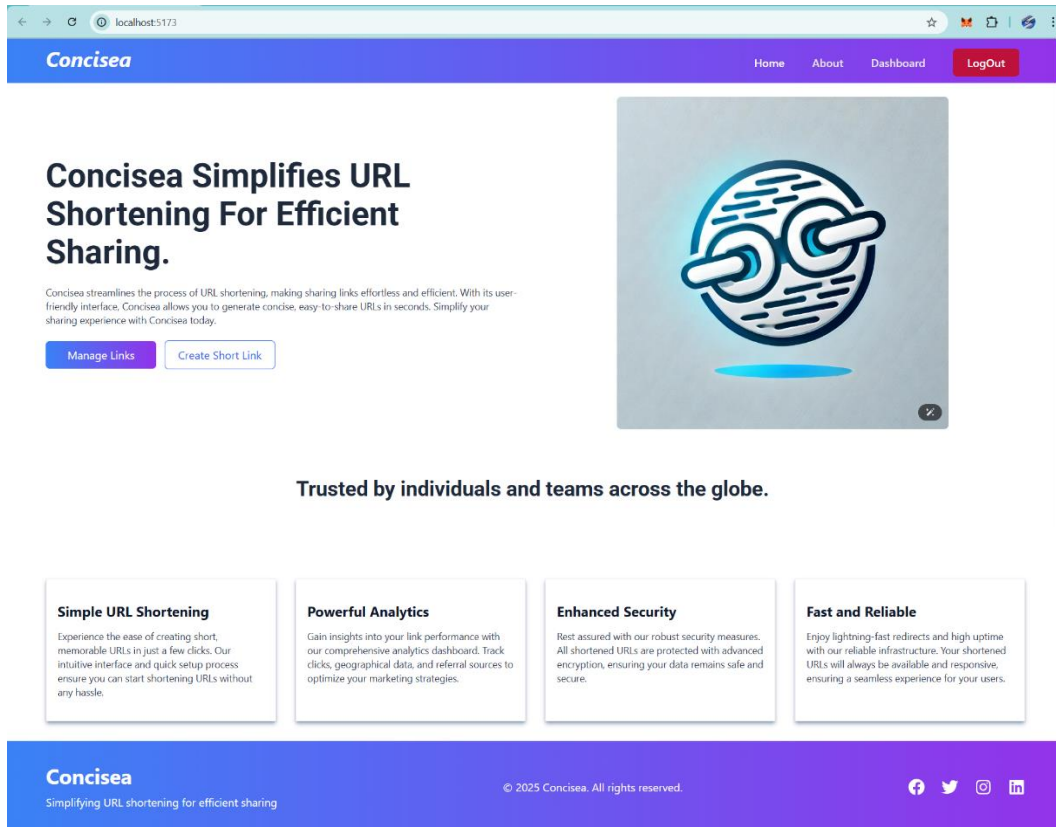
Example Code Snippet for Data Validation:

```
const urlSchema = new mongoose.Schema({
  originalUrl: { type: String, required: true },
  shortUrl: { type: String, required: true, unique: true },
  createdAt: { type: Date, default: Date.now }
});
```

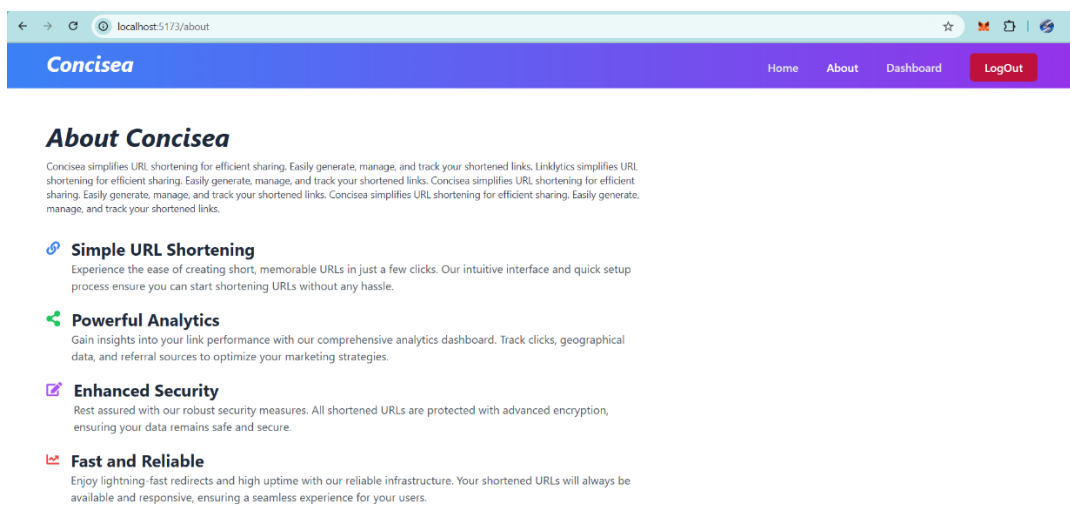
By implementing thorough error handling, rigorous testing, and strong data validation mechanisms, the project ensures reliability, security, and a seamless user experience.

CHAPTER 9: OUTPUTS AND RESULTS

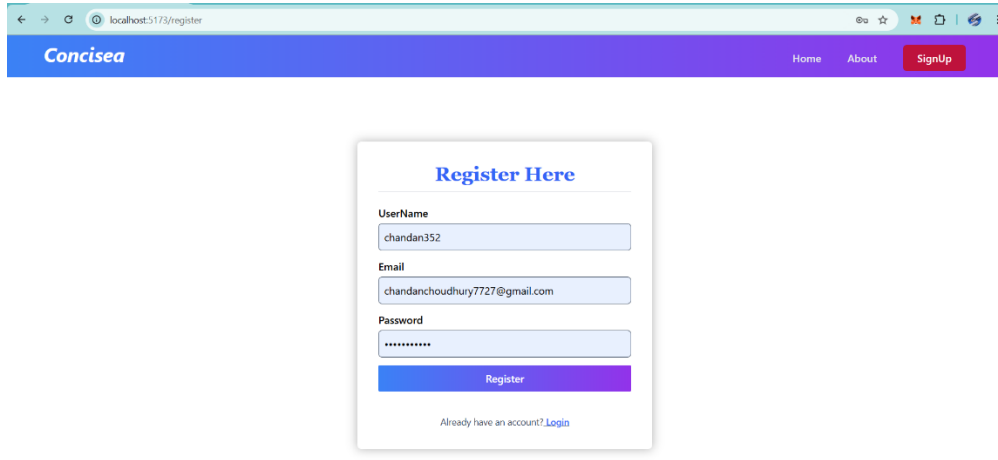
Home Page



About page

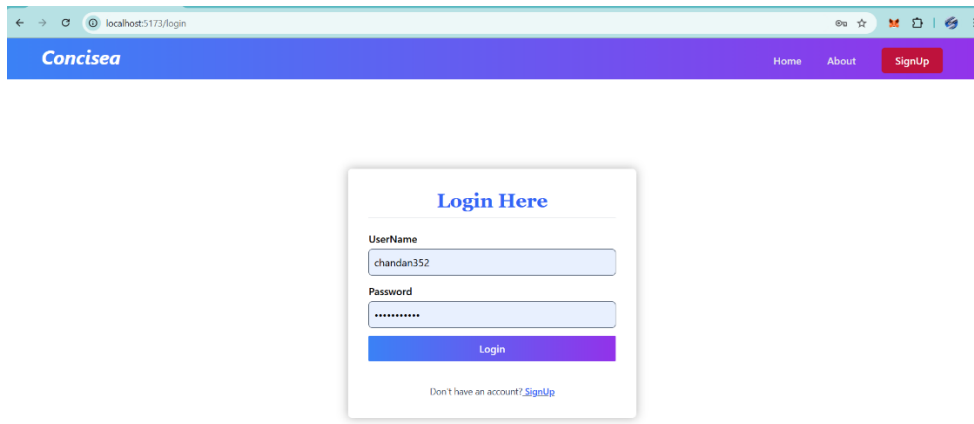


Signup page



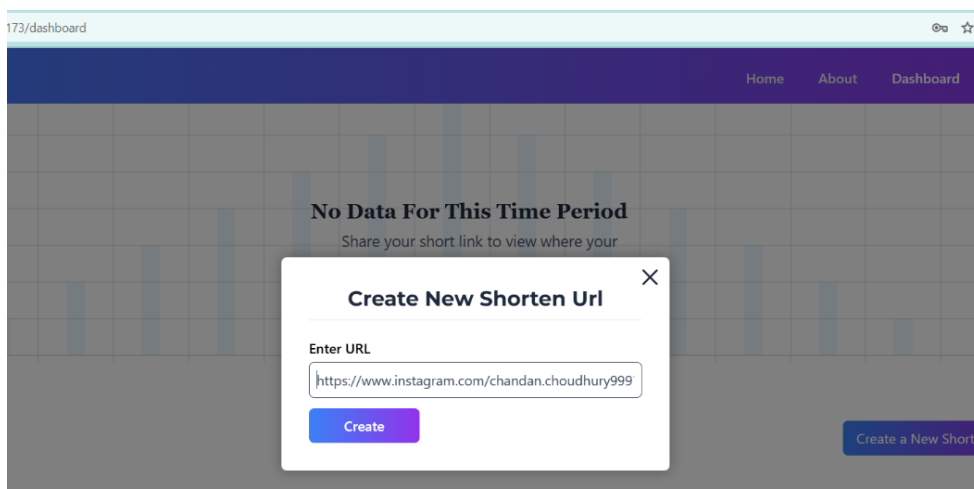
The screenshot shows a web browser at localhost:5173/register. The page has a blue header with the 'Concisea' logo and navigation links for 'Home', 'About', and 'SignUp'. The main content area features a white 'Register Here' form. The form includes input fields for 'UserName' (filled with 'chandan352'), 'Email' (filled with 'chandan.choudhury7727@gmail.com'), and 'Password' (masked with dots). A blue 'Register' button is at the bottom of the form, and a link for 'Already have an account? Login' is below it.

Login page



The screenshot shows a web browser at localhost:5173/login. The page has a blue header with the 'Concisea' logo and navigation links for 'Home', 'About', and 'SignUp'. The main content area features a white 'Login Here' form. The form includes input fields for 'UserName' (filled with 'chandan352') and 'Password' (masked with dots). A blue 'Login' button is at the bottom of the form, and a link for 'Don't have an account? SignUp' is below it.

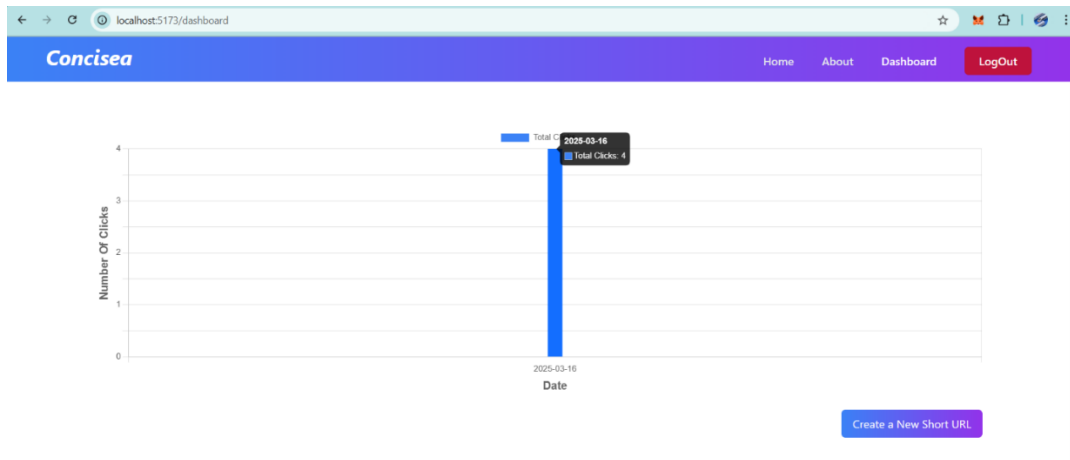
Dashboard (To Enter your URL and Create a Short URL)



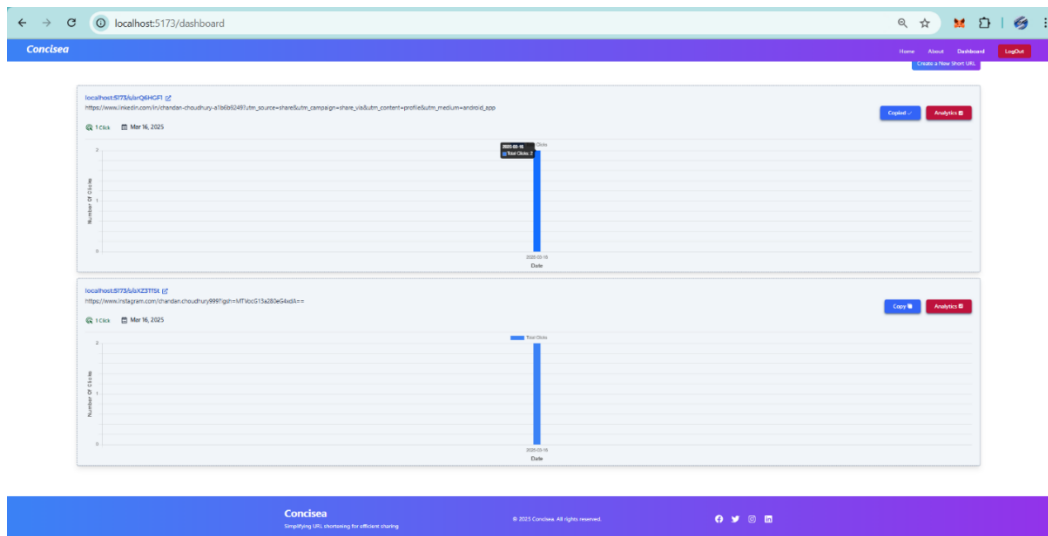
The screenshot shows a web browser at localhost:5173/dashboard. The page has a blue header with the 'Concisea' logo and navigation links for 'Home', 'About', and 'Dashboard'. The main content area features a grey background with a grid pattern. A modal dialog box is open in the center, titled 'Create New Shorten Url'. The modal has a close button (X) in the top right corner. It contains an input field labeled 'Enter URL' with the text 'https://www.instagram.com/chandan.choudhury999'. Below the input field is a blue 'Create' button. In the bottom right corner of the dashboard, there is a blue button labeled 'Create a New Short U'.

Dashboard (Analytics)

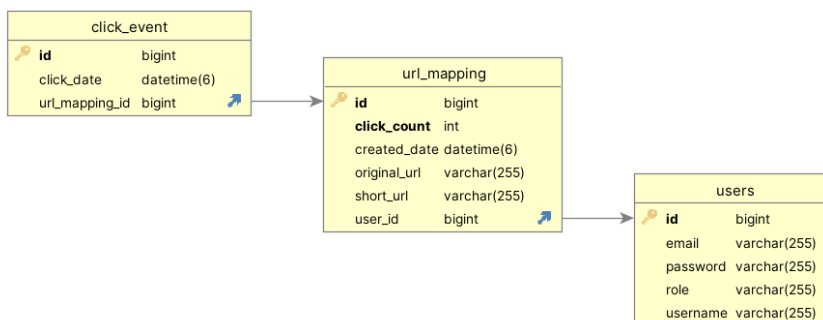
Bar graph Analytics of All the links you Created



Individual Analytics of the links you created



Schema of the Database



CHAPTER 10: FUTURE IMPLEMENTATIONS

- **Hosting the Website**

- Deploy the project on cloud platforms like **Netlify (for frontend), and Render or AWS (for backend)** to make it publicly accessible.
- Use **Docker** for containerization to ensure a consistent deployment environment.
- Set up **CI/CD pipelines** for automated deployment.

- **Forgot Password Functionality**

- Implement a **password reset feature** where users receive a password reset link via email.
- Use **JWT-based or OTP-based password reset authentication** for security.
- Store **hashed security tokens** in the database for password recovery.

- **User Role Management**

- Add **Admin and Regular User roles** to control access and permissions.
- Admins can manage all shortened URLs, while regular users can only view their own.

- **Custom Short URLs**

- Allow users to **create custom aliases** instead of system-generated URLs (e.g., `https://short.ly/mycustomurl`).
- Validate if the alias is already taken and suggest alternatives.

- **QR Code Generation**

- Provide a **QR code** for each shortened URL to make it easy for users to scan and access.
- Use **Google Charts API or Python libraries** like `qrcode` for QR code generation.

- **URL Expiry and Auto-Deletion**

- Implement **URL expiration dates** where users can set a time limit after which the URL becomes inactive.
- Set up a **cron job** to periodically delete expired URLs.

- **Analytics and Insights (Advanced Features)**

- Display **geolocation insights** to see where users are clicking from.
- Capture which website the user came from before clicking the link (Facebook, Twitter, Google, LinkedIn, etc.).

CHAPTER 11: CONCLUSION

The URL Shortener project successfully provides a fast, efficient, and user-friendly platform for generating and managing short links. By implementing a robust backend with MongoDB, Express.js, and Node.js, along with a responsive React and Tailwind CSS frontend, the system ensures seamless URL shortening, redirection, and basic analytics tracking.

The project effectively addresses key functionalities such as **secure user authentication, real-time link management, and click tracking**, allowing users to monitor the performance of their shortened URLs. Additionally, the integration of a **dashboard** enhances usability by providing insights into URL activity.

While the project achieves its primary objectives, there are several areas for future improvements. Features such as **password reset functionality, advanced analytics, link expiration settings, and hosting the application online** can significantly enhance usability and security. Moreover, integrating **QR code generation, custom short URLs, and API access for developers** can further expand the project's scope.

Overall, the URL Shortener project demonstrates a well-structured and scalable approach to modern web development. With additional enhancements, it can evolve into a comprehensive link management solution suitable for personal, business, and enterprise use.

REFERENCES

• Official Documentation

- Spring Boot: <https://spring.io>
- ReactJS: <https://react.dev>
- Tailwind CSS: <https://tailwindcss.com>

• Online Resources

- PostgreSQL Docs: <https://www.postgresql.org/docs/>
- JWT Authentication: <https://jwt.io>

Project Contributions

- Stack Overflow & GitHub discussions on URL shortening in Java.
- YouTube Tutorials for Database Setup in Cloud.

GitHub Link of Our Project: https://github.com/ChandanChoudhury7727/Concisea_-Bitly-Clone-.git