How to Think Like a Developer: Building a URL Shortener from Scratch

Table of Contents

The Developer's Mindset

How Developers Think

Developers don't just start coding. They think in **systems** and **problems**. Here's the mental process:

1. **Understand the Problem** 🤔
   - What exactly does the user want?
   - What are the core features?
   - What are the edge cases?
2. **Break It Down** 🧩
   - Split big problems into smaller ones
   - Identify what needs to be built
   - Figure out dependencies
3. **Plan Before Code** 📝
   - Design the structure
   - Choose technologies
   - Think about data flow
4. **Build Incrementally** 🔨
   - Start with the simplest version
   - Add features one by one
   - Test as you go

The "Why" Behind Every Decision

When you see code, ask yourself:

- **Why is this file here?**
- **Why is this component separate?**
- **Why use this library?**
- **Why structure it this way?**

Problem Analysis

Step 1: Understanding the Problem

Question: "I want to build a URL shortener like bit.ly"

Developer's Mental Process:

```
🤔 What does a URL shortener actually do?
   → Takes long URLs and makes them short
   → When someone clicks short URL, redirects to original
   → Should track how many times it's clicked

🤔 Who will use this?
   → People who want to share links
   → People who want to track link performance
   → Marketing teams, social media users

🤔 What features do they need?
   → Create short URLs
   → View their URLs
   → See click statistics
   → Maybe customize short URLs
   → Account to save their URLs
```

Step 2: Core Features Identification

Must-Have Features (MVP - Minimum Viable Product):

1. Shorten a URL
2. Redirect short URL to original
3. Basic click tracking

Nice-to-Have Features:

1. User accounts
2. Dashboard to manage URLs
3. Analytics graphs
4. Custom short URLs
5. URL expiration

Advanced Features:

1. QR codes
2. Password protection
3. Geographic analytics
4. API for developers

Step 3: User Stories

Developers think in "user stories" - what the user wants to accomplish:

```
As a user, I want to:
- Enter a long URL and get a short one
- Share the short URL with others
```

```
- See how many people clicked my link
- Manage all my shortened URLs in one place
- Create an account to save my URLs
```

Planning Phase

Step 1: Draw the User Flow

Before any code, sketch out what the user will do:

```
User Journey:
1. Visit website
2. [Optional] Register/Login
3. Enter long URL
4. Get short URL
5. Share short URL
6. View analytics

Technical Flow:
1. Frontend receives long URL
2. Send to backend API
3. Backend generates short code
4. Store in database
5. Return short URL to frontend
6. When short URL clicked:
   - Backend looks up original URL
   - Increments click counter
   - Redirects to original URL
```

Step 2: Data Structure Planning

What data do we need to store?

```
// URL Object
{
  id: "unique-id",
  originalUrl: "https://very-long-url.com/with/many/parameters",
  shortCode: "abc123",
  userId: "user-id",
  clickCount: 42,
  createdDate: "2024-01-01",
  clicks: [
    { date: "2024-01-01", count: 10 },
    { date: "2024-01-02", count: 15 }
  ]
}

// User Object
{
  id: "user-id",
  username: "john_doe",
```

```
  email: "john@example.com",
  password: "hashed-password",
  createdDate: "2024-01-01"
}
```

Step 3: API Endpoints Planning

What endpoints do we need?

```
Authentication:
POST /api/auth/register - Create account
POST /api/auth/login    - Login user

URL Management:
POST /api/urls/shorten     - Create short URL
GET  /api/urls/myurls      - Get user's URLs
GET  /api/urls/:shortCode  - Redirect to original URL

Analytics:
GET /api/urls/analytics/:shortCode - Get click data for specific URL
GET /api/urls/totalClicks          - Get overall click statistics
```

Architecture Decisions

Frontend Architecture Thinking

Question: "How should I organize my React app?"

Developer's Thought Process:

```
🤔 What pages do I need?
   → Landing page (homepage)
   → Login/Register pages
   → Dashboard (for logged-in users)
   → About page
   → Error page

🤔 What components will I reuse?
   → Input fields (for forms)
   → Buttons
   → Navigation bar
   → Footer
   → URL cards/items

🤔 How will I handle navigation?
   → React Router for different pages
   → Some pages need authentication
   → Need to protect certain routes

🤔 How will I manage state?
   → User authentication status (global)
```

```
   → Form data (local)
   → API data (server state)
```

Folder Structure Decision

```
src/
├── components/          # Reusable UI components
│   ├── Dashboard/      # Dashboard-specific components
│   ├── TextField.jsx   # Reusable input component
│   ├── NavBar.jsx      # Navigation
│   └── ...
├── pages/              # Full page components
├── hooks/              # Custom React hooks
├── api/                # API communication
├── context/            # Global state
└── utils/              # Helper functions
```

Why this structure?

- **Separation of Concerns**: Each folder has a specific purpose
- **Reusability**: Components can be used anywhere
- **Maintainability**: Easy to find and update code
- **Scalability**: Easy to add new features

State Management Decisions

Question: "How should I manage state?"

Developer's Analysis:

```
🤔 What state do I have?
   1. User authentication (global - needed everywhere)
   2. Form inputs (local - only in that component)
   3. API data (server state - needs caching)
   4. UI state (local - modals, toggles)

🤔 What tools should I use?
   → Context API: For authentication (simple global state)
   → React Query: For API data (caching, loading states)
   → useState: For local component state
   → React Hook Form: For form state
```

Why these choices?

- **Context API**: Simple, built into React, perfect for auth
- **React Query**: Handles API complexity (loading, errors, caching)
- **Local State**: Keeps components simple and focused

Feature Breakdown

Feature 1: URL Shortening

Developer's Planning:

```
🎯 Goal: User enters long URL, gets short URL

📝 Requirements:
- Input field for long URL
- Validation (must be valid URL)
- API call to backend
- Display generated short URL
- Copy to clipboard functionality

🔧 Components needed:
- Input component with validation
- Submit button
- Result display
- Copy button

💾 Data flow:
User Input → Validation → API Call → Store Result → Display
```

Feature 2: User Dashboard

Developer's Planning:

```
🎯 Goal: Show user's URLs and analytics

📝 Requirements:
- List all user's shortened URLs
- Show click counts
- Display creation dates
- Analytics graphs
- Create new URLs

🔧 Components needed:
- Dashboard layout
- URL list component
- Individual URL item
- Analytics graph
- Create URL modal

💾 Data flow:
Load Dashboard → Fetch User URLs → Display List → Click Analytics → Show Graph
```

Feature 3: Authentication

Developer's Planning:

```
🎯 Goal: Users can create accounts and login

📝 Requirements:
- Registration form
- Login form
- Password validation
- Token storage
```

```
- Route protection

🔧 Components needed:
- Login form
- Register form
- Private route wrapper
- Navigation updates

💾 Data flow:
Form Submit → API Call → Store Token → Update UI → Protect Routes
```

Technology Selection

Why React?

Developer's Reasoning:

```
✅ Component-based architecture (reusable pieces)
✅ Large ecosystem (many libraries available)
✅ Virtual DOM (fast updates)
✅ Strong community support
✅ Good for single-page applications
```

Why These Specific Libraries?

React Router DOM

Problem: Need to show different pages without full page reloads **Solution**: Client-side routing

```jsx
// Instead of separate HTML files, we have:
<Route path="/" element={<LandingPage />} />
<Route path="/dashboard" element={<Dashboard />} />
```

React Query

Problem: API calls are complex (loading, errors, caching) **Solution**: Library that handles all API complexity

```jsx
// Instead of manual useState/useEffect:
const { data, isLoading, error } = useQuery('urls', fetchUrls);
```

Context API

Problem: Authentication state needed in many components **Solution**: Global state management

```jsx
// Instead of passing props through many levels:
const { token } = useStoreContext(); // Available anywhere
```

Tailwind CSS

Problem: Writing custom CSS is time-consuming **Solution**: Utility classes for rapid styling

```
// Instead of writing CSS files:
<div className="bg-blue-500 text-white p-4 rounded-lg">
```

Development Workflow

Phase 1: Setup and Basic Structure

Developer's Approach:

1. **Create Project**

```
npm create vite@latest url-shortener --template react
cd url-shortener
npm install
```

2. **Install Dependencies**

```
npm install react-router-dom axios react-query tailwindcss
```

3. **Create Basic Structure**

```
src/
├── components/
├── pages/
├── hooks/
├── api/
└── context/
```

Phase 2: Core Functionality

Step-by-Step Building:

1. **Start with Static Pages**
   - Create basic components without functionality
   - Focus on layout and design
   - No API calls yet
2. **Add Routing**
   - Set up React Router
   - Create navigation between pages
   - Test that routing works
3. **Add Authentication**
   - Create login/register forms
   - Set up Context API
   - Add route protection
4. **Connect to Backend**

- Set up Axios
- Create API functions
- Add React Query

5. **Add URL Shortening**
   - Create URL input form
   - Connect to backend API
   - Display results
6. **Add Dashboard**
   - Fetch user's URLs
   - Display in list
   - Add analytics

Phase 3: Polish and Features

1. **Styling and UX**
2. **Error handling**
3. **Loading states**
4. **Animations**
5. **Responsive design**

Testing Your Thinking

Before You Code, Ask Yourself:

1. Component Planning

```
🤔 What does this component do?
🤔 What data does it need?
🤔 What can the user do with it?
🤔 How does it connect to other components?
```

2. State Planning

```
🤔 What data changes over time?
🤔 Which components need this data?
🤔 Should this be local or global state?
🤔 How do I update this data?
```

3. API Planning

```
🤔 What data do I need from the backend?
🤔 When should I fetch this data?
🤔 How do I handle loading and errors?
🤔 Should I cache this data?
```

Common Beginner Mistakes to Avoid

❌ Mistake 1: Starting with Code

```
Bad: "Let me start coding the login form"
Good: "Let me plan what the login form needs to do"
```

❌ Mistake 2: Building Everything at Once

```
Bad: "Let me build the entire dashboard with all features"
Good: "Let me build a simple dashboard that just shows 'Hello'"
```

❌ Mistake 3: Not Planning State

```
Bad: "I'll figure out state management later"
Good: "Let me plan what data I need and where it should live"
```

❌ Mistake 4: Ignoring User Experience

```
Bad: "The functionality works, that's enough"
Good: "How does this feel from the user's perspective?"
```

The Creator's Thought Process

When the Original Developer Built This App:

Step 1: Requirements Gathering

```
💭 "I need to build a URL shortener. What does that mean?"
   → Users paste long URLs
   → System generates short codes
   → Short URLs redirect to original URLs
   → Users want to track clicks
   → Users want to manage their URLs
```

Step 2: Technical Planning

```
💭 "What technology should I use?"
   → Frontend: React (good for interactive UIs)
   → Backend: Node.js/Express (JavaScript everywhere)
   → Database: MongoDB/PostgreSQL (store URLs and users)
   → Styling: Tailwind (fast development)
```

Step 3: Architecture Design

```
💭 "How should this work technically?"
   → Frontend sends long URL to backend
   → Backend generates random short code
   → Store mapping in database
   → Return short URL to frontend
   → When short URL accessed, lookup and redirect
```

Step 4: User Experience Design

```
💭 "What should the user see?"
   → Simple homepage explaining the service
   → Clear form to enter URLs
   → Dashboard to manage URLs
   → Analytics to see performance
   → Easy copy/share functionality
```

Step 5: Implementation Strategy

```
💭 "What should I build first?"
   1. Basic React app with routing
   2. Simple URL shortening (no auth)
   3. Add user authentication
   4. Add dashboard and management
   5. Add analytics and graphs
   6. Polish UI and add animations
```

Why They Made Specific Choices

Choice: React Context API for Authentication

Thinking:

```
💭 "I need to know if user is logged in throughout the app"
💭 "Context API is simple and built into React"
💭 "I don't need complex state management for just auth"
```

Choice: React Query for API Calls

Thinking:

```
💭 "API calls have loading states, errors, and need caching"
💭 "React Query handles all this complexity for me"
💭 "I don't want to write useEffect for every API call"
```

Choice: React Router for Navigation

Thinking:

```
💭 "I want different pages without page reloads"
💭 "React Router is the standard for React apps"
💭 "I need protected routes for dashboard"
```

Choice: Tailwind CSS for Styling

Thinking:

```
💭 "I want to build UI quickly without writing custom CSS"
💭 "Tailwind gives me utility classes for everything"
💭 "It's easier to make responsive designs"
```

How to Start Building

Phase 1: Foundation (Week 1)

Goal: Get basic React app running

Tasks:

1. Create new React project with Vite
2. Set up Tailwind CSS
3. Create basic folder structure
4. Build static homepage (no functionality)
5. Add React Router with basic pages

Learning Focus:

- How React components work
- How to structure a React project
- Basic JSX and styling

Phase 2: Authentication (Week 2)

Goal: Users can register and login

Tasks:

1. Create login/register forms
2. Set up Context API for auth state
3. Connect forms to backend API
4. Add route protection
5. Handle login/logout

Learning Focus:

- React forms and validation
- API calls with Axios
- Global state management
- Route protection

Phase 3: Core Functionality (Week 3)

Goal: URL shortening works

Tasks:

1. Create URL input form
2. Connect to backend shortening API
3. Display generated short URLs
4. Add copy to clipboard
5. Handle errors and loading

Learning Focus:

- Form handling
- API integration
- Error handling
- User feedback

Phase 4: Dashboard (Week 4)

Goal: Users can manage their URLs

Tasks:

1. Create dashboard layout
2. Fetch and display user's URLs
3. Add URL management features
4. Create individual URL components
5. Add basic analytics

Learning Focus:

- Data fetching with React Query
- Component composition
- State management
- List rendering

Phase 5: Analytics (Week 5)

Goal: Visual analytics with charts

Tasks:

1. Set up Chart.js
2. Create graph components
3. Fetch analytics data
4. Display click trends
5. Add interactive features

Learning Focus:

- Third-party library integration
- Data visualization
- Advanced React patterns

Phase 6: Polish (Week 6)

Goal: Professional-looking app

Tasks:

1. Add animations with Framer Motion
2. Improve responsive design
3. Add loading states everywhere
4. Improve error handling
5. Add notifications

Learning Focus:

- Animation libraries
- User experience
- Performance optimization
- Professional polish

Decision-Making Framework

When Facing Any Coding Decision, Ask:

1. Functionality Questions

- What exactly should this do?
- What are the edge cases?
- How should errors be handled?

2. User Experience Questions

- Is this intuitive for users?
- What happens when they click this?
- How do they know what to do next?

3. Technical Questions

- Where should this logic live?
- How does this connect to other parts?
- Is this the simplest solution?

4. Maintenance Questions

- Will I understand this code in 6 months?
- Is this easy to modify later?
- Are there too many dependencies?

Example Decision Process

Scenario: "Should I put the URL shortening form on the homepage or separate page?"

Developer's Analysis:

```
🤔 User Experience:
   → Users want to shorten URLs immediately
   → Homepage should show main functionality
   → But logged-in users might want dashboard instead

🤔 Technical Considerations:
   → Homepage needs to handle both auth states
   → Form component should be reusable
   → Need to handle success/error states

🤔 Decision:
   → Put simple form on homepage for quick access
   → Also have full form in dashboard for logged-in users
   → Make form component reusable
```

React-Specific Thinking

Component Design Philosophy

1. Single Responsibility

Each component should do ONE thing well:

```
// Good: TextField only handles input
function TextField({ label, value, onChange, error }) {
  return (
    <div>
      <label>{label}</label>
      <input value={value} onChange={onChange} />
      {error && <span>{error}</span>}
    </div>
  );
}

// Bad: TextField also handles API calls
function TextField({ label, apiEndpoint, userId }) {
  // Too many responsibilities!
}
```

2. Data Flow

Think about how data moves:

```
Parent Component (has state)
    ↓ (passes data via props)
Child Component (displays data)
    ↓ (calls function via props)
Parent Component (updates state)
```

3. When to Create New Components

Create a new component when:

- You're repeating similar JSX
- A component is getting too large (>100 lines)
- You need to reuse functionality
- You want to separate concerns

Hook Usage Philosophy

useState: For Local Component State

```
const [isOpen, setIsOpen] = useState(false); // Modal open/closed
const [loading, setLoading] = useState(false); // Button loading state
```

useEffect: For Side Effects

```
useEffect(() => {
  // Fetch data when component mounts
  fetchUserData();
}, []); // Empty dependency array = run once

useEffect(() => {
  // Update document title when data changes
```

```
  document.title = `${data.length} URLs`;
}, [data]); // Runs when 'data' changes
```

Custom Hooks: For Reusable Logic

```
function useAuth() {
  const [user, setUser] = useState(null);
  const login = (credentials) => { /* login logic */ };
  const logout = () => { /* logout logic */ };
  return { user, login, logout };
}
```

Common Development Patterns

Pattern 1: Loading States

```
function MyComponent() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetchData()
      .then(setData)
      .catch(setError)
      .finally(() => setLoading(false));
  }, []);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;
  return <div>{/* render data */}</div>;
}
```

Pattern 2: Form Handling

```
function MyForm() {
  const [formData, setFormData] = useState({ name: '', email: '' });

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

  const handleSubmit = (event) => {
    event.preventDefault();
    // Submit form data
```

```
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={formData.name}
        onChange={handleChange('name')}
      />
      {/* ... */}
    </form>
  );
}
```

Pattern 3: Conditional Rendering

```
function Dashboard() {
  const { user } = useAuth();
  const { data: urls, loading } = useQuery('urls', fetchUrls);

  return (
    <div>
      {loading && <Spinner />}
      {!loading && urls.length === 0 && <EmptyState />}
      {!loading && urls.length > 0 && <UrlList urls={urls} />}
    </div>
  );
}
```

Building Mindset

Start Small, Think Big

```
Week 1: "Can I show a list of hardcoded URLs?"
Week 2: "Can I add a new URL to the list?"
Week 3: "Can I save URLs to a backend?"
Week 4: "Can I add user accounts?"
Week 5: "Can I add analytics?"
```

Always Ask "What's the Simplest Version?"

```
Complex: "Build a dashboard with graphs, filters, and export"
Simple: "Show a list of URLs with click counts"
Start: "Show a hardcoded list of URLs"
```

Think in User Stories

```
Instead of: "I need to build a form component"
Think: "As a user, I want to enter a URL and get a short version"
```

Embrace Iteration

```
Version 1: Basic functionality, ugly UI
Version 2: Add styling and better UX
Version 3: Add advanced features
Version 4: Optimize and polish
```

Next Steps for You

1. Study the Existing Code

- Read through each file in the PDF guide
- Try to understand WHY each piece exists
- Trace the data flow from user action to result

2. Build Your Own Version

- Start with a simple static page
- Add one feature at a time
- Don't worry about making it perfect

3. Practice the Mindset

- Before coding, always plan
- Break big problems into small ones
- Ask "What's the simplest version?"
- Focus on user experience

4. Learn by Doing

- Copy parts of the existing code
- Modify them to understand how they work
- Break things and fix them
- Experiment with different approaches

Remember: **Every expert was once a beginner.** The key is to start simple and build up your understanding gradually. Don't try to understand everything at once - focus on one concept at a time and build from there.

The developer who built this app didn't start with the complex version you see. They built it piece by piece, making decisions and learning along the way. You can do the same!