**Design Choices in Full-Stack Ticketing Application**

**a) API Endpoint Design and Business Logic Mapping**

The backend RESTful service follows a resource-oriented architecture with endpoint naming conventions that directly map to the core business entities and operations of the ticketing system:

**Ticket Resource:**

* GET /api/tickets - Retrieves all tickets

With optional query parameters for filtering: status, priority, reporterId

* POST /api/tickets - Creates a new ticket
* GET /api/tickets/{id} - Retrieves a specific ticket
* PUT /api/tickets/{id} - Updates a ticket's details
* DELETE /api/tickets/{id} - Removes a ticket

**Comment Resource:**

* GET /api/tickets/{id}/comments - Retrieves all comments for a specific ticket
* POST /api/tickets/{id}/comments - Adds a comment to a specific ticket

**User Resource:**

* GET /api/users - Retrieves all users

**Error Handling**: All API calls include error handling through the errorHandlingService

**Query Parameters for Filtering**: The endpoints support query parameters like status, priority, and assignee to filter tickets based on business needs. For example:

* GET /api/tickets?status=open&priority=high

This allows stakeholders to quickly identify high-priority issues that need immediate attention.

**Business Logic Mapping**:

Each endpoint maps directly to a Spring service layer that encapsulates business rules. For example, the ticket creation endpoint enforces required fields (title, description) and sets default values (creation date, status). The assignment endpoint performs validation to ensure only valid users can be assigned tickets

**b) Frontend Component and Route Structure**

The Angular frontend architecture was carefully designed with these considerations:

1. **Feature-Based Organization**:

Components are organized by feature (tickets, users, comments). This structure aligns with business domains rather than technical concerns

1. **Routing Strategy**:

|  |
| --- |
| const routes: Routes = [  // Default route - redirects to login  {  path: '',  redirectTo: '/login',  pathMatch: 'full',  },    // Authentication  {  path: 'login',  component: LoginComponent,  },    // Main dashboard with ticket list  {  path: 'dashboard',  component: DashboardComponent, // Contains TicketListComponent  },    // Ticket operations  {  path: 'ticket/:id',  component: TicketDetailComponent,  },  {  path: 'create-ticket',  component: CreateTicketComponent,  },    // Wildcard route for 404 handling  {  path: '\*\*',  redirectTo: '/login',  },  ]; |

Top-level routes map to standalone pages (LoginComponent, DashboardComponent, etc.). Child components (TicketListComponent, TicketCardComponent) are embedded within their parents rather than having their own routesIt supports hierarchical navigation from list views to detailed views

**c) Specific Bindings and Directives Usage**

The Application Leverages Angular's Binding Mechanisms and Directives

1. Two-Way Binding in Forms:

|  |
| --- |
| <input   type="text"   id="title"  [(ngModel)]="newTicket.title"  name="title"  required  placeholder="Enter ticket title"/> |

Used in create and edit ticket forms for real-time data synchronization. This allows for immediate validation and feedback during data entry.

2. Structural Directives for Conditional Display:

|  |
| --- |
| @if (isLoading) {      <div class="loading">Loading tickets...</div>    } @else {      <!-- Feature 6: Conditional for empty state -->      @if (tickets.length === 0) {        <div class="empty-state">          <p>No tickets found</p>        </div>      } @else {        <!-- List view of tickets -->  } |

Uses Angular's modern @if syntax to conditionally show loading states or content. Implements empty state handling with @if (tickets.length === 0)

3. Class Binding for Dynamic Styling:

|  |
| --- |
| <div class="ticket-list-item" [class]="'priority-' + ticket.priority.toLowerCase()" (click)="ticket.id && onTicketClick(ticket.id)"> |

Applies a visual indicator through dynamic class binding for different ticket properties. Creates a consistent color-coding system for priorities (red for high, orange for medium, green for low). This would enhance usability by allowing users to quickly identify important or urgent tickets just based on the color.

**d) HTTP Client Call Placement**

The TicketListComponent is responsible for retrieving and displaying a collection of tickets, and it appropriately makes its own HTTP calls for several strategic reasons. Since this component owns the presentation of the ticket list, it naturally takes responsibility for fetching the data it displays. It also manages its own loading state, allowing it to control when and how loading indicators appear during data retrieval. In addition, because the component supports user-initiated filtering, it needs to handle HTTP requests that include query parameters directly. The component also performs data transformation by adding tickets with reporter usernames before displaying them, so that the API data is converted into presentation ready information while keeping a clean separation between data logic and display.

The TicketDetailComponent, on the other hand, is designed to manage a single resource, which is an individual ticket and its related comments. It performs HTTP calls to retrieve data for a specific ticket based on the ID provided through route parameters. This makes sense because the component’s responsibility aligns with detail-level operations such as viewing, updating, or deleting a ticket. By performing these HTTP calls in its ngOnInit() lifecycle method, the component ensures that data is automatically loaded whenever the user navigates to a particular ticket. This design keeps the component self-contained and independent, avoiding the need for parent components to pass down complete ticket data.

The CreateTicketComponent focuses on the creation of new tickets and appropriately performs its own HTTP POST operations. As the owner of the ticket creation form, it has direct access to all form values required for the API call. Handling these operations within the component also confines error handling to the same context where the user is interacting, allowing for precise feedback through success or failure messages. Additionally, the component can manage navigation after successful ticket creation, such as redirecting back to the ticket list. This approach maintains a clean separation of concerns by preventing the need to emit form data to parent components for processing.

**e) Reactive Forms Design and Validation**

Unfortunately, I didn’t use any reactive forms within this project due to time constraints.

**f) Integration of Frontend and Backend for End-User Experience**

All these design decisions work together to create a cohesive end-user experience:

1. **Consistent Mental Model**:
   * The API structure, component organization, and routing all reinforce the same conceptual model of tickets, users, and workflows
   * This consistency reduces cognitive load on users
2. **Real-Time Feedback Loop**:
   * Form validation provides immediate feedback on user input
   * HTTP responses trigger appropriate UI updates (success messages, error notifications)
   * Loading indicators keep users informed during asynchronous operations
3. **Business Logic Reinforcement**:
   * Frontend components visualize business states (ticket statuses, priorities)
   * UI elements adapt to user roles and permissions
   * Navigation paths guide users through proper workflow sequences
4. **Performance Considerations**:
   * Lazy loading improves initial page load
   * Strategic data fetching minimizes network traffic
   * Optimistic UI updates make the application feel responsive
5. **Accessibility and Usability**:
   * Form labels and error messages help all users interact successfully
   * Visual cues (colors, icons) reinforce status information
   * Keyboard navigation supports efficient operation

The end result is a ticketing system where the frontend provides an intuitive interface that accurately represents the business domain modeled in the backend. Users can efficiently create, track, and resolve tickets while the system enforces necessary business rules and data integrity at both the UI and API levels.