Juice Shop – Application Design Doc

SEC353 supporting material

Overview

Juice Shop is an online e-commerce store for organic, high-quality, fruit & vegetable juice products. The main of goal Juice Shop is to provide customers across the globe access to our delicious fruit & vegetable juice products, in addition to other categories of Juice Shop-branded merchandise such as one-of-a-kind artwork, swag (such as shirts, stickers, temporary tattoos, 3D-printed logos), digital content (like a Juice Shop-themed DLC for Table Top Simulator), and more!

As the storefront for our business, the Juice Shop e-commerce platform must provide all the relevant functionality that customers may require. This includes account management, product discovery and detail pages, shopping cart functionality with a checkout flow, and internationalization support to allow customers across the globe to purchase items.

Juice Shop must also be built and deployed using technology that meets our security, availability, performance, and monitoring standards.

Functional Requirements

In this section we'll detail the specific requirements that the Juice Shop e-commerce platform must satisfy to be a viable product for our customers, broken down by functional section:

Shopping Experience

- Customers must be able to browse a paginated table of Juice Shop items for sale, with each item having its own detail page. An item includes a title, a picture, description, and price. All items support customer-submitted reviews.
- Juice Shop must support an item search feature to allow customers to find a particular item by title
- Juice Shop must support a shopping basked experience, where customers can add one or more items to their basket prior to checkout

Account Management

- Juice Shop must support customer accounts. Customers can only add-to-cart when they are logged in, not as unauthenticated users. Support for login and logout is required.
- Juice Shop must support email & password accounts, as well as Google social login
- A Juice Shop user profile must support a custom username, and a profile picture - either uploaded directly or linked via URL
- A Juice Shop account must make available to the user: their order history, saved addresses, payment options (credit card, digital wallets)

Security & privacy features

- Juice Shop must allow customers to request a complete data erasure
- Juice Shop must allow customers to export the user-specific data that the site has stored
- Juice Shop must support 2FA, password reset, and a screen to show the last location the user has accessed the site from

Customer feedback for Juice Shop

- Juice Shop must allow customers to provide feedback to the Juice Shop team, this should support free text submissions and/or a numeric rating
- Customers must be able to access a Photo Wall page on the site, showing off real-world photos shared by other users of Juice Shop products in public
- Customers must be able to interact with a Juice Shop chatbot for real-time question and answer about the site and their account
- Customers must be able to view on the site details about the history of Juice Shop, that includes recent feedback from other users and links to our social pages.

High-Level System Architecture

Juice Shop will be architected using a modern, microservices-inspired approach while maintaining simplicity of deployment and operational overhead. The system will be structured in layers, with components built through APIs to interface other components.

The main three components will be:

1. A frontend web application that is responsible for rendering Juice Shop, and navigates the user through the various flows described in the Functional Requirements section

2. A REST API layer that powers the web application. This API layer implements the business logic to facilitate customers' usage of the web application.
3. A SQLite database and file storage solution that will hold customers' information, item listings, and other information for the Juice Shop store.

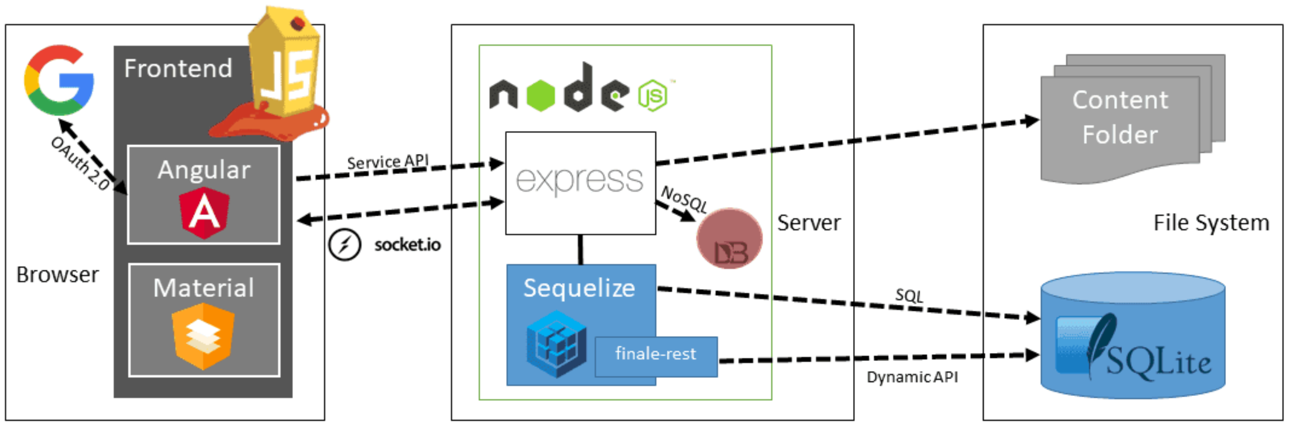
Technology Stack

Frontend: Juice Shop's frontend application will be implemented using Angular & Material, a batteries-included JavaScript framework for building complex single page applications (SPA) with a modern UI supported out of the box.

API components: JavaScript is used exclusively in the backend API services. An Express application will be built and hosted via Node.js to deliver a RESTful to the frontend.

Data storage: For data persistence, Juice Shop will deploy a SQLite database to store all relational data. This includes info for users, products, payments, shopping carts, etc. A Sequelize ORM will be used to manage access to SQLite from the Express application. Additionally, a MarsDB instance will be added to store user reviews and order information in a NoSQL structure. Third, Juice Shop will host an FTP server as a convenient method to store company information, including planned business decisions, private customer incident information, and metadata on deployed software packages for maintainers to reference.

Juice Shop uses SQLite3 as its persistence layer with Sequelize ORM for data modeling and query abstraction. The database file is stored at `data/juiceshop.sqlite` and is created on first application startup.



API Application Design

Juice Shop is a Node.js application built on Express. It's architected as a monolithic application with individual APIs for business functions and clear separation of concerns in implementation.

Entry Point Flow

```
app.ts → server.ts → start() → HTTP server listening
```

Key Bootstrap Sequence:

1. **Initialization** (`server.ts`)
   - Loads middleware libraries (cors, rate-limiting, etc.)
   - Cleanup FTP file folder
2. **Database Initialization** (`models/index.ts`)
   - Creates Sequelize instance with SQLite
   - Initializes 20 models in specific order
   - Establishes relationships via `relationsInit()`
3. **Data Seeding** (`data/datacreator.ts`)
   - Runs `sequelize.sync({ force: true })` - drops and recreates all tables
   - Seeds data in dependency order:

     ```
     SecurityQuestions → Users → Challenges → Products →
     Baskets → BasketItems → Feedback → Complaints → etc.
     ```
4. **Server Start**

- Binds to port (default 3000)

Request Processing Pipeline

```
Incoming Request
     ↓
1. CORS (allow all origins)
     ↓
2. Metrics Collection (Prometheus)
     ↓
3. Body Parsing (urlencoded → text → custom JSON)
     ↓
4. Access Logging (rotating file)
```

```
     ↓
5. Rate Limiting (selective endpoints)
     ↓
6. Authentication
     ↓
7. Authorization (custom middleware)
     ↓
8. Route Handler
     ↓
9. Error Handler
     ↓
Response
```

Authentication & Authorization Design

1. **Route-level** (applied to entire route groups):

   ```
   app.use('/rest/basket', security.isAuthorized(), security.appendUserId())
   app.use('/api/BasketItems', security.isAuthorized())
   ```

2. **Method-level** (HTTP verb restrictions):

   ```
   app.post('/api/Products', security.isAuthorized())
   app.delete('/api/Products/:id', security.denyAll())
   ```

3. **Role-based** (accounting, admin):

   ```
   app.use('/api/Quantitys/:id',
     security.isAccounting(),
     IpFilter(['123.456.789'], { mode: 'allow' })
   )
   ```

Access logging

Access logging will be enabled for the Express application using the `winston` library, a popular utility for web applications. Each API in Juice Shop will use a centralized logger to record events to local, rotated files. These logs will contain information about the request made, the response status code, and any errors that occurred during processing.

Sensitive information such as customer secrets, payment information, and Personally Identifiable Information (PII) will be redacted using a specialized `winston` log formatter, or by using the supported "rewriter" functionality.

`DEBUG` and `INFO` logging will not be supported in production environments, preventing noise and protecting internal application state from being viewed through logging events.

Logging will cover the following security-related events for the application:

1. All basic API activity including purchases, feedback submission and User profile changes
2. Monitor access to administrative operations that include sensitive data
3. Monitor login and logout access patterns, like high rate of authorization failures (example threshold: > 5 attempts per User)
4. Monitor customer requests for data exports

Each log event will include the following fields to support further investigation: User identity, timestamp of access, source IP address, and metadata on request outcome.

Logs will be hardened against tampering by appending a fingerprint with each log event. Only team members with the special `Operator` role will be permitted to access log files. Out-of-date log files will be promptly deleted upon rotation to prevent accidental modification of prior data.

Access logs will be retained for at least one year, accessible only by the security team through a ticketed process. Logs will be monitored actively by the team and connected to automated alarms for detection of anomalous activity (example scenario: many login attempts from different IP addresses). Based on testing, there is confirmation this logging setup provides the necessary context for investigating operational and security events.

Database Access Patterns

1. **Sequelize ORM**:

   ```
   UserModel.findOne({ where: { email: req.body.email } })
   ```

2. **Raw SQL**:

   ```
   models.sequelize.query(
     `SELECT * FROM Users WHERE email = '${req.body.email}'
      AND password = '${security.hash(req.body.password)}'`
   )
   ```

3. **NoSQL** for reviews and other document content:

   ```
   // data/nosql.ts
   reviewsCollection.insert({ productId, message, author })
   ```

API Generation Strategy

Finale-REST (auto-generates CRUD endpoints):

```
const autoModels = [
  { name: 'User', exclude: ['password', 'totpSecret'], model: UserModel },
  { name: 'Product', exclude: [], model: ProductModel },
```

```
  // ... 14 models total
]

for (const { name, exclude, model } of autoModels) {
  finale.resource({
    model,
    endpoints: [`/api/${name}s`, `/api/${name}s/:id`],
    excludeAttributes: exclude,
    pagination: false
  })
}
```

Generated endpoints:

- `GET /api/Users` - list all users
- `GET /api/Users/:id` - get user by ID
- `POST /api/Users` - create user
- `PUT /api/Users/:id` - update user
- `DELETE /api/Users/:id` - delete user
- etc.**

Custom REST Routes

Pattern: Functional route handlers

```
// routes/login.ts
export function login() {
  return (req: Request, res: Response, next: NextFunction) => {
    // Login logic
  }
}

// server.ts
app.post('/rest/user/login', login())
```

Route organization:

- `/api/*` - Auto-generated CRUD (Finale)
- `/rest/*` - Custom business logic

Internationalization (i18n)

Translation Strategy

Backend i18n:

```
i18n.configure({
  locales: ['en', 'de', 'es', 'fr', ...], // 40+ languages
  directory: path.resolve('i18n'),
  cookie: 'language',
  defaultLocale: 'en',
  autoReload: true
})

// Usage in routes
res.send(res.__('Invalid email or password.'))
```

Frontend i18n:

- Angular i18n with JSON files in `frontend/src/assets/i18n/`
- Dynamic loading based on user preference

On-the-fly translation (Finale hooks):

```
resource.list.fetch.after((req, res, context) => {
  for (let i = 0; i < context.instance.length; i++) {
    context.instance[i].description =
req.__(context.instance[i].description)
  }
})
```

Monitoring & Observability

Prometheus Metrics

Startup metrics:

```
const startupGauge = new Prometheus.Gauge({
  name: `${appName}_startup_duration_seconds`,
  help: 'Duration required to perform startup tasks',
  labelNames: ['task']
})
```

```
// Measure async operations
const end = startupGauge.startTimer({ task: 'datacreator' })
await datacreator()
end()
```

Request metrics:

```
app.use(metrics.observeRequestMetricsMiddleware())
app.use(metrics.observeFileUploadMetricsMiddleware())
```

Metrics endpoint:

```
app.get('/metrics', metrics.serveMetrics()) // Public metrics endpoint
```

Frontend Design

The Juice Shop frontend is an Angular 20 single-page application. The application demonstrates modern Angular patterns including standalone components, reactive programming with RxJS, and Material Design integration

Design Principles

The frontend architecture follows several key principles that guide implementation decisions:

Separation of Concerns: Components handle presentation and user interaction, while services manage business logic, HTTP communication, and state.

Reactive Programming : We embrace RxJS observables throughout the application rather than promises or callbacks. This provides consistent async handling, powerful composition operators, and built-in cancellation support.

Application Bootstrap and Initialization

The application uses Angular's `bootstrapApplication` function. All providers, including services, guards, and configuration, are registered directly in the bootstrap configuration.

Environment Detection: First, we check if we're running in production mode and enable optimizations accordingly. This affects things like change detection strategy, error handling verbosity, and source map generation.

HTTP Interceptor Registration: A global HTTP interceptor is registered to automatically inject JWTs (signature optional for testing) into all outgoing requests. This
centralizes authentication header management and eliminates the need for services to manually handle auth tokens.

Internationalization Setup: The `ngx-translate` module is configured with an HTTP loader that fetches translation files on demand. This allows us to support 40+ languages without loading all translations upfront.

Service Registration: All services are registered as singleton providers at the root level. This ensures a single instance of each service exists
throughout the application lifecycle, maintaining consistent state.

Route Guard Registration: Authentication and authorization guards are registered to protect routes based on user login status and roles

Route Organization

Routes are organized into four categories based on access requirements:

Public Routes: Accessible to all users without authentication. These include the product search, login, registration, and informational pages.
The default route redirects to the search page, making product browsing the primary entry point.

Authenticated Routes: Require a JWT. Allow unsigned JWTs for testing, and in production allow HMAC 256 or RSA 256 algorithms. These include the shopping cart, checkout flow, order history, and user profile pages. The LoginGuard checks for token existence before allowing access.

Role-Based Routes: Require specific user roles in addition to authentication. The administration page requires admin role, and the accounting page requires accounting role. These use specialized guards that decode the JWT and check the role claim.

Route Guard Implementation

Route guards implement Angular's CanActivate interface to control route access. The guard system has a hierarchical structure where more specific
guards depend on more general ones.

LoginGuard is the base guard that checks for a JWT token. If no token exists, it redirects to a 403 error page with an appropriate error message. It also provides a tokenDecode method that other guards use to extract the JWT payload.

AdminGuard and AccountingGuard build on LoginGuard by additionally checking the user's role claim in the JWT payload. They use LoginGuard's token decoding functionality and add role-specific validation.

DeluxeGuard is slightly different—it's not a route guard but rather a service that components can inject to check deluxe membership status. This
allows components to conditionally show features based on membership level.

Core Services

UserService handles all user-related operations including authentication, registration, password management, and profile updates. It maintains an
isLoggedIn subject that broadcasts login state changes, allowing components throughout the application to react to authentication events.

BasketService manages the shopping cart, including adding items, updating quantities, removing items, and checkout. It maintains an itemTotal
subject that broadcasts the current cart item count, which the navbar displays. The service coordinates with the backend to ensure cart state
persists across sessions.

ProductService handles product catalog operations including search, filtering, and retrieval. It communicates with both REST endpoints for
product data and NoSQL endpoints for product reviews, demonstrating the application's polyglot persistence approach.

Database Design

Sequelize Configuration

Location: `models/index.ts`

```
typescript
const sequelize = new Sequelize('database', 'username', 'password', {
  dialect: 'sqlite',
  retry: {
    match: [/SQLITE_BUSY/],
    name: 'query',
    max: 5
  },
  transactionType: Transaction.TYPES.IMMEDIATE,
  storage: 'data/juiceshop.sqlite',
  logging: false
})
```

Key Configuration Decisions:

- **Immediate transactions:** Prevents write conflicts by acquiring locks immediately
- **Retry logic:** Handles SQLite's `SQLITE_BUSY` errors (common in concurrent scenarios) with up to 5 retries
- **File-based storage:** Single-file database makes deployment and backup trivial

Database Schema

Core Entity Tables

1. Users (Authentication & Authorization)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  username: STRING (default: '')
  email: STRING UNIQUE
  password: STRING (bcrypt)
  role: STRING (customer|deluxe|accounting|admin)
  deluxeToken: STRING
  lastLoginIp: STRING (default: '0.0.0.0')
  profileImage: STRING
  totpSecret: STRING (for 2FA)
  isActive: BOOLEAN (default: true)
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
  deletedAt: TIMESTAMP (soft delete via paranoid mode)
}
```

Design Notes:

- **Paranoid mode enabled:** Soft deletes preserve user data for audit trails
- **Password setter:** Automatically hashes passwords using bcrypt
- **Role validation:** Enforces enum constraint at ORM level
- **Profile image logic:** Admin role automatically gets special default image
- **Ephemeral accountant hook:** `afterValidate` hook prevents direct creation of accountant user

2. Products (Catalog)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  name: STRING
  description: STRING
  price: DECIMAL
  deluxePrice: DECIMAL
  image: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
  deletedAt: TIMESTAMP (paranoid)
}
```

Design Notes:

- **Dual pricing:** Regular and deluxe member pricing
- **Paranoid mode:** Products can be "discontinued" without losing historical data

3. Baskets (Shopping Cart)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users)
  coupon: STRING (nullable)
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **One basket per user:** Enforced at application level via `findOrCreate`
- **Coupon field:** Stores applied discount codes

4. BasketItems (Join Table)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  BasketId: INTEGER (FK to Baskets, non-updatable)
  ProductId: INTEGER (FK to Products, non-updatable)
  quantity: INTEGER
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **Many-to-many relationship:** Links Baskets and Products
- **Non-updatable foreign keys:** `makeKeyNonUpdatable()` prevents FK manipulation after creation
- **Quantity tracking:** Allows multiple units of same product

User-Related Tables

5. Addresses

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  fullName: STRING
  mobileNum: INTEGER (validation: 1000000-9999999999)
  zipCode: STRING (length: 1-8)
  streetAddress: STRING (length: 1-160)
  city: STRING
  state: STRING (nullable)
  country: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **One-to-many:** Users can have multiple addresses
- **Validation at ORM level:** Phone number range, zip code length, address length

6. Credit Cards (Payment Methods)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  fullName: STRING
  cardNum: INTEGER (validation: 16 digits)
  expMonth: INTEGER (1-12)
  expYear: INTEGER (2080-2099)
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **Data format:** Stores credit card numbers as integers
- **Future-dated expiry:** Year validation 2080-2099
- **No encryption:** Card data stored in plaintext

7. Wallets (Digital Currency)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  balance: INTEGER (default: 0)
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **One wallet per user:** Enforced at application level

Feedback & Support Tables

8. Feedback items

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, NOT constrained)
  comment: STRING
  rating: INTEGER
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **Anonymous feedback allowed:** No FK constraint, `UserId` can be null

9. Complaints

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  message: STRING
  file: STRING (file path)
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **File attachment support:** Stores file path, not binary data
- **User-linked:** Requires authenticated user

Security Tables

10. SecurityQuestions

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  question: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

11. SecurityAnswers

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, NOT constrained)
  SecurityQuestionId: INTEGER (FK to SecurityQuestions, constrained)
  answer: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Design Notes:

- **Password recovery mechanism:** Links users to security questions

Auxiliary Tables

12. Quantities (Inventory)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  ProductId: INTEGER (FK to Products, constrained)
  quantity: INTEGER
  limitPerUser: INTEGER (nullable)
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

13. Recycles (Recycling Program)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  AddressId: INTEGER (FK to Addresses, constrained)
  quantity: INTEGER
  isPickup: BOOLEAN
  date: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

14. Photo Wall

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  imagePath: STRING
  caption: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

15. Captchas (Bot Protection)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  captchaId: INTEGER
  captcha: STRING
  answer: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

16. ImageCaptchas

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, NOT constrained)
  image: STRING
  answer: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

17. PrivacyRequests (GDPR)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  UserId: INTEGER (FK to Users, constrained)
  deletionRequested: BOOLEAN
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

18. Deliveries (Shipping Methods)

```
{
  id: INTEGER PRIMARY KEY AUTOINCREMENT
  name: STRING
  price: DECIMAL
  deluxePrice: DECIMAL
  eta: INTEGER
  icon: STRING
  createdAt: TIMESTAMP
  updatedAt: TIMESTAMP
}
```

Relationship Design

Defined in: `models/relations.ts`

One-to-Many Relationships

`User → Addresses` (constrained)
`User → Cards` (constrained)
`User → Baskets` (constrained)
`User → Complaints` (constrained)
`User → Feedback items` (NOT constrained - allows anonymous)
`User → Memories` (constrained)
`User → PrivacyRequests` (constrained)
`User → Recycles` (constrained)

`User → SecurityAnswers` (NOT constrained)
`User → Wallets` (constrained)
`User → ImageCaptchas` (NOT constrained)
`Product → Quantities` (constrained)
`SecurityQuestion → SecurityAnswers` (constrained)
`Address → Recycles` (constrained)

Many-to-Many Relationships

`Basket ↔ Product` (through BasketItems)**