Bootcamp NAO x TecMilenio

Campus Monterrey

GROUP 1

Java Spark for web apps

Sofía Pérez Guzmán

NAO ID: 3350

10/27/25

Backlog

Sprint 1 — API and basic configuration

As a **System Administrator**, I want to configure the Maven project and Spark framework, so that the application builds, runs, and exposes HTTP endpoints for development and testing.

- Requirements:
    - Maven pom.xml with dependencies for JDK, Spark, Gson/Jackson, logging, WebSocket support, and testing (JUnit).
    - Standard project structure (src/main/java, src/main/resources, src/test).
    - Application entry point that starts Spark with a configurable port via environment variable or application.properties.
    - Logging setup and a health-check endpoint (/health) that returns 200 OK and JSON status.
    - Build scripts or Maven goals to compile, run tests, and start the app.

User story 2

As an **API client**, I want CRUD endpoints for user accounts, so that I can create, read, update, and delete users programmatically.

- Requirements:
    - Routes:
        - POST /api/users — create user.
        - GET /api/users — list users with optional pagination query params (page, size).
        - GET /api/users/:id — get user by id.
        - PUT /api/users/:id — update user.
        - DELETE /api/users/:id — delete user.
    - Request/response format: JSON; consistent envelope with status and data.
    - User model: id (UUID), username (string), email (string), role (string: buyer|seller|admin), createdAt (ISO8601).

    - Validation: username (required, 3–30 chars), email (required, valid format), role constrained.
    - Appropriate HTTP codes: 201 for create, 200 for success, 204 for delete, 400 for validation errors, 404 for not found.
    - Unit tests for each endpoint and validation scenarios.

User story 3

As an **API client**, I want CRUD endpoints for collectible items, so that I can manage items available in the store.

- Requirements:
    - Routes:
        - POST /api/items — create item (seller or admin).
        - GET /api/items — list items with optional filters (category, minPrice, maxPrice, status, query).
        - GET /api/items/:id — get item details.
        - PUT /api/items/:id — update item.
        - DELETE /api/items/:id — delete item.
    - Item model: id (UUID), title, description, category, price (decimal), currency, stock (int), sellerId (UUID), status (AVAILABLE|SOLD|HIDDEN), createdAt, updatedAt.
    - Validation: title (required), price (>=0), stock (>=0), category (required).
    - Pagination for list endpoint; consistent JSON responses and HTTP codes as above.
    - Unit and integration tests for standard flows and error cases.

User story 4

As a **Developer**, I want centralized error handling and JSON error payloads, so that API clients receive consistent, actionable error messages.

- Requirements:
    - Global exception handler that maps exceptions to HTTP responses.
    - Error response shape: { errorCode, message, details? }.

    - Handle validation errors (400), resource not found (404), conflict (409), internal errors (500).
    - Log errors with stack traces for server-side logs; avoid revealing internal details in responses.

Sprint 2 — Views, templates, forms, and exception handling

User story 5

As a **Shopper**, I want a public item listing page, so that I can browse available collectibles.

- Requirements:
    - Mustache templates for pages.
    - Route GET /items renders items view populated server-side.
    - Server-side supports applying same filters used by API; HTML lists items with title, thumbnail, price, status.
    - Responsive basic layout (HTML/CSS) and accessible markup (alt attributes, headings).
    - Server provides necessary model data to template (items, pagination info, filter state).

User story 6

As a **Seller**, I want an item creation form in the UI, so that I can submit offers to list new collectibles.

- Requirements:
    - GET /items/new renders form template; POST /items handles form submission.
    - Form fields: title, description, category (select), price, currency, stock, images (optional URL), tags.
    - Server-side validation with clear UI error messages shown in template.
    - On success, redirect to item details page with success flash message.
    - CSRF protection for form submissions.

User story 7

As a **Shopper**, I want an item detail page, so that I can see full information and contact or purchase options.

- Requirements:
    - GET /items/:id renders detail template with images, full description, seller info, price, stock, and status.

    - Action buttons depend on status and user role (Buy, Make Offer, Edit for owner).
    - Graceful handling and user-friendly error page when item not found or unavailable.

User story 8

As a **Developer**, I want robust form validation and user-friendly error handling, so that user input problems are clearly communicated.

- Requirements:
    - Server-side validators with field-level error messages returned to template.
    - Client-side basic validation for better UX (non-authoritative).
    - Exception handling route for template rendering errors that returns a neat error page (500) and logs details.

Sprint 3 — Filters, security, and real-time updates

User story 9

As a **Shopper**, I want to filter and sort items, so that I can quickly find collectibles matching my preferences.

- Requirements:
    - UI controls for filters: category, price range, availability, seller, tags, text search.
    - Sort options: price asc/desc, newest, most popular.
    - Server-side endpoints must accept filter and sort query params and return filtered, paginated results.
    - Apply filters both for API GET /api/items and server-rendered GET /items, producing identical results.
    - Ensure safe input handling to prevent injection attacks.

User story 10

As a **Site Administrator**, I want request filters (middleware) applied centrally, so that common logic is enforced consistently.

- Requirements:
    - Implement Spark filters or before/after handlers for:
        - Logging requests and responses (basic info).
        - Authentication checks for protected routes.
        - CORS configuration for API endpoints.
        - Common headers (Content-Type, cache-control).

    - Filters should be configurable and have minimal latency impact.

User story 11

As a **Shopper**, I want real-time price updates on item listing and detail pages, so that I always see the latest price without refreshing.

- Requirements:
    - WebSocket endpoint (e.g., /ws/prices) to broadcast price updates.
    - Server-side logic to publish price changes when an item is updated (PUT /api/items/:id or internal price-change events).
    - Client-side JS to open WebSocket, subscribe to item channels or receive update messages, and update DOM price fields in real time.
    - Message format: JSON with itemId, newPrice, currency, timestamp.
    - Handle connection lifecycle events (reconnect, error) gracefully in client code.

User story 12

As a **Developer**, I want data validation and optimistic concurrency for price updates, so that simultaneous edits don't cause inconsistent state.

- Requirements:
    - Item updates include an optimistic concurrency token (version or updatedAt).
    - PUT /api/items/:id rejects updates with stale tokens with 409 Conflict and explanatory error.
    - WebSocket broadcasts reflect committed updates only.

User story 13

As a **Security Officer**, I want authentication and role-based access control, so that actions like creating/editing/deleting items are limited to authorized users.

- Requirements:
    - Simple authentication mechanism (session-based or JWT) configurable for the exercise.
    - Protected routes: POST/PUT/DELETE for items and user management require appropriate role.
    - Middleware should enforce RBAC and return 401/403 as appropriate.

    - Passwords stored hashed for any persisted credentials in the demo environment.

Final submission — Integration, documentation, and presentation

User story 14

As a **Project Evaluator**, I want a packaged final project with source, build instructions, PDF analysis, and MP4 presentation, so that I can review design, implementation, and results.

- Requirements:

  Final repository including README with setup and run instructions, architecture overview, API docs, and sample data seeding steps.

  PDF report (analysis and results) describing architecture, choices, endpoints, security measures, test coverage, and known limitations.

  MP4 video (screen + narration) demoing the app: setup, key features (CRUD, forms, filters, WebSockets), and tests running.

  Automated tests: unit tests and at least basic integration tests for API routes; test results and code coverage summary included.

Cross-cutting non-functional and acceptance criteria

Acceptance criteria

- All API endpoints pass automated tests that cover success and error scenarios.
- UI templates render without server errors and display validation messages.
- WebSocket updates appear in the client within 2 seconds of server commit during normal operation.
- Filters and middleware do not increase average request latency by more than a reasonable test threshold for the exercise.
- The final submission ZIP/repo includes the PDF and MP4 and instructions to reproduce locally.

Non-functional requirements

- Use Java 11+ and Spark framework.
- Use Mustache templating for server-rendered views.
- JSON serialization using Gson or Jackson.

- Persist data in-memory for the demo; provide an optional simple persistence adapter (H2 or file-based) if desired.
- Clear, consistent logging with different levels (INFO, WARN, ERROR).
- Provide seed script or sample data loader.

Testing and quality

- Unit tests for models, validation, and utilities.
- Integration tests for key API flows (users, items, price update).
- Manual test plan for UI flows including form submission, error cases, and WebSocket updates.

Sprint 1— API and basic configuration

User Story	Requirements
As a System Administrator, I want to configure the Maven project and Spark framework, so that the application builds, runs, and exposes HTTP endpoints for development and testing.	• Maven pom.xml with dependencies for JDK, Spark, Gson/Jackson, logging, WebSocket support, and testing (JUnit). • Standard project structure (src/main/java, src/main/resources, src/test). • Application entry point that starts Spark with a configurable port via environment variable or application.properties.. • Logging setup and a health-check endpoint (/health) that returns 200 OK and JSON status. • Build scripts or Maven goals to compile, run tests, and start the app.

As an API client, I want CRUD endpoints for user accounts, so that I can create, read, update, and delete users programmatically.	Routes: • POST /api/users — create user. • GET /api/users — list users with optional pagination query params (page, size). • GET /api/users/:id — get user by id. • PUT /api/users/:id — update user. • DELETE /api/users/:id — delete user. • Request/response format: JSON; consistent envelope with status and data. • User model: id (UUID), username (string), email (string), role (string: buyer|seller|admin), createdAt (ISO8601). • Validation: username (required, 3–30 chars), email (required, valid format), role constrained. • Appropriate HTTP codes: 201 for create, 200 for success, 204 for delete, 400 for validation errors, 404 for not found. • Unit tests for each endpoint and validation scenarios.

As an API client, I want CRUD endpoints for collectible items, so that I can manage items available in the store.	Routes: • POST /api/items — create item (seller or admin). • GET /api/items — list items with optional filters (category, minPrice, maxPrice, status, query). • GET /api/items/:id — get item details. • PUT /api/items/:id — update item. • DELETE /api/items/:id — delete item. • Item model: id (UUID), title, description, category, price (decimal), currency, stock (int), sellerId (UUID), status (AVAILABLE|SOLD|HIDDEN), createdAt, updatedAt. • Validation: title (required), price (>=0), stock (>=0), category (required). • Pagination for list endpoint; consistent JSON responses and HTTP codes as above. • Unit and integration tests for standard flows and error cases.

As a Developer, I want centralized error handling and JSON error payloads, so that API clients receive consistent, actionable error messages.	• Global exception handler that maps exceptions to HTTP responses. • Error response shape: { errorCode, message, details? }. • Handle validation errors (400), resource not found (404), conflict (409), internal errors (500). • Log errors with stack traces for server-side logs; avoid revealing internal details in responses.

Sprint 2 — Views, templates, forms, and exception handling

User Story	Requirements
As a Shopper, I want a public item listing page, so that I can browse available collectibles.	• Mustache templates for pages. • Route GET /items renders items view populated server-side. • Server-side supports applying same filters used by API; HTML lists items with title, thumbnail, price, status. • Responsive basic layout (HTML/CSS) and accessible markup (alt attributes, headings). • Server provides necessary model data to template (items, pagination info, filter state).
As a Seller, I want an item creation form in the UI, so that I can submit offers to list new collectibles.	• GET /items/new renders form template; POST /items handles form submission. • Form fields: title, description, category (select), price, currency, stock, images (optional URL), tags. • Server-side validation with clear UI error messages shown in template. • On success, redirect to item details page with success flash message. • CSRF protection for form submissions.

As a Shopper, I want an item detail page, so that I can see full information and contact or purchase options.	• GET /items/:id renders detail template with images, full description, seller info, price, stock, and status. • Action buttons depend on status and user role (Buy, Make Offer, Edit for owner). • Graceful handling and user-friendly error page when item not found or unavailable.
As a Developer, I want robust form validation and user-friendly error handling, so that user input problems are clearly communicated.	• Server-side validators with field-level error messages returned to template. • Client-side basic validation for better UX (non-authoritative). • Exception handling route for template rendering errors that returns a neat error page (500) and logs details.

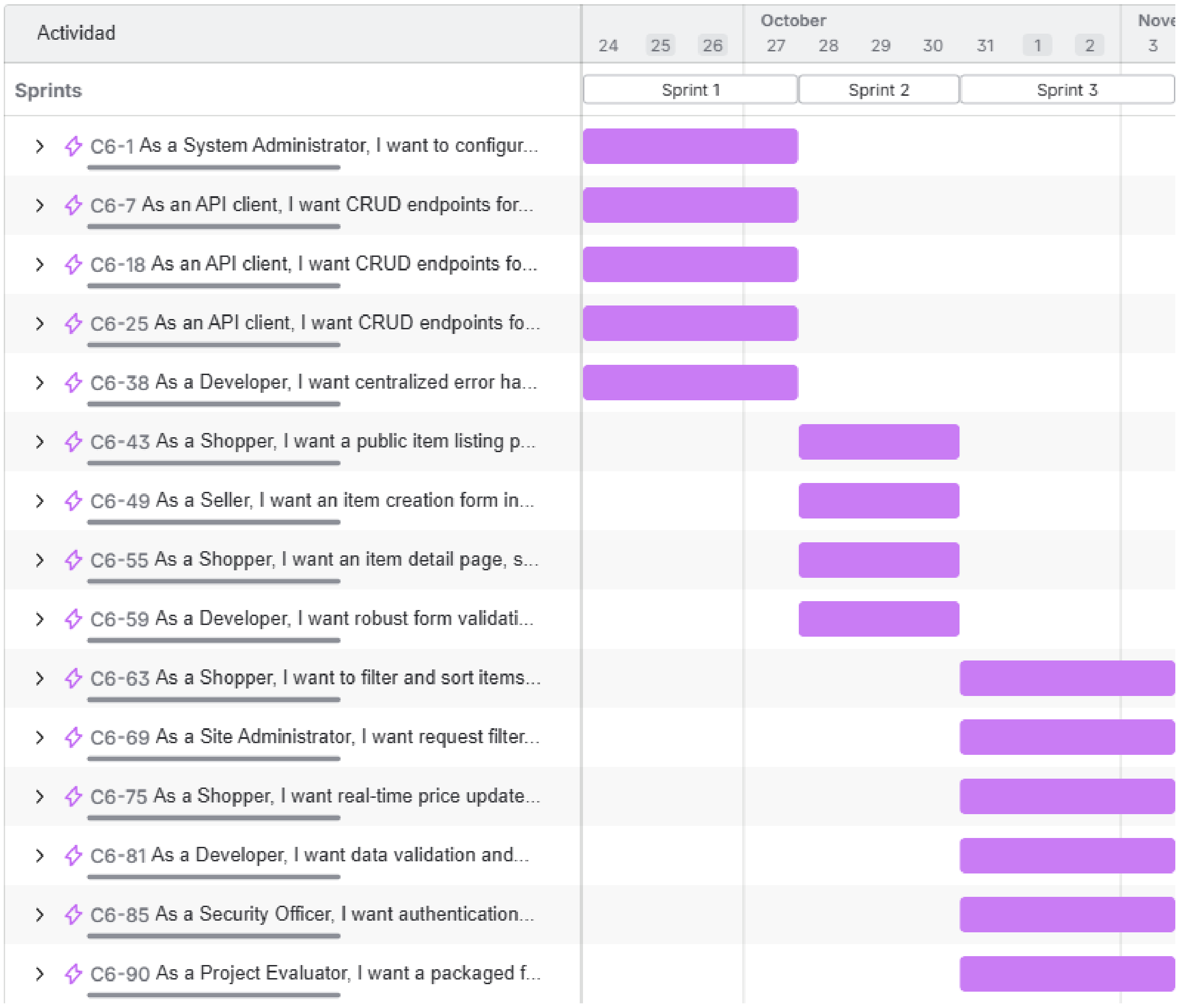
Sprint 3 — Filters, security, and real-time updates

User Story	Requirements
As a Shopper, I want to filter and sort items, so that I can quickly find collectibles matching my preferences.	• UI controls for filters: category, price range, availability, seller, tags, text search. • Sort options: price asc/desc, newest, most popular. • Server-side endpoints must accept filter and sort query params and return filtered, paginated results. • Apply filters both for API GET /api/items and server-rendered GET /items, producing identical results. • Ensure safe input handling to prevent injection attacks.
As a Site Administrator, I want request filters (middleware) applied centrally, so that common logic is enforced consistently.	• Implement Spark filters or before/after handlers for: • Logging requests and responses (basic info). • Authentication checks for protected routes. • CORS configuration for API endpoints.

	• Common headers (Content-Type, cache-control). • Filters should be configurable and have minimal latency impact.
As a Shopper, I want real-time price updates on item listing and detail pages, so that I always see the latest price without refreshing.	• WebSocket endpoint (e.g., /ws/prices) to broadcast price updates. • Server-side logic to publish price changes when an item is updated (PUT /api/items/:id or internal price-change events). • Client-side JS to open WebSocket, subscribe to item channels or receive update messages, and update DOM price fields in real time. • Message format: JSON with itemId, newPrice, currency, timestamp. • Handle connection lifecycle events (reconnect, error) gracefully in client code.
As a Developer, I want data validation and optimistic concurrency for price updates, so that simultaneous edits don't cause inconsistent state.	• Item updates include an optimistic concurrency token (version or updatedAt). • PUT /api/items/:id rejects updates with stale tokens with 409 Conflict and explanatory error. • WebSocket broadcasts reflect committed updates only.
As a Security Officer, I want authentication and role-based access control, so that actions like creating/editing/deleting items are limited to authorized users.	• Simple authentication mechanism (session-based or JWT) configurable for the exercise. • Protected routes: POST/PUT/DELETE for items and user management require appropriate role. • Middleware should enforce RBAC and return 401/403 as appropriate. • Passwords stored hashed for any persisted credentials in the demo environment.

Roadmap

https://sofipguz.atlassian.net/jira/software/projects/C6/boards/167/backlog?ignoreStickyVersion=true&atlOrigin=eyJpIjoiMGE0ZWE5NmMwMTgzNGY3ZDhiNWUzMWYwZjI5M2YyMmIiLCJwIjoiaiJ9



C6-1 As a System Administrator, I want to configur...
C6-7 As an API client, I want CRUD endpoints for...
C6-18 As an API client, I want CRUD endpoints fo...
C6-25 As an API client, I want CRUD endpoints fo...
C6-38 As a Developer, I want centralized error ha...
C6-43 As a Shopper, I want a public item listing p...
C6-49 As a Seller, I want an item creation form in...
C6-55 As a Shopper, I want an item detail page, s...
C6-59 As a Developer, I want robust form validati...
C6-63 As a Shopper, I want to filter and sort items...
C6-69 As a Site Administrator, I want request filter...
C6-75 As a Shopper, I want real-time price update...
C6-81 As a Developer, I want data validation and...
C6-85 As a Security Officer, I want authentication...
C6-90 As a Project Evaluator, I want a packaged f...