PROJECT ASSIGNMENT GUIDELINES

J. R. Almeida, I. C. Oliveira, L. Bastião, J. M. Fernandes

v2023-10-16

Index

1 Project objectives

1.1 Project assignment objectives

The project assignment will require that students apply selected software engineering practices, including:

- Develop a product specification, from the usage scenarios (user stories) to the technical design.
- Propose, justify, and implement a software architecture, based on enterprise *frameworks*.
- Apply collaborative work practices, both in code development and agile project management.

1.2 Team and roles

Each team/group should assign the roles described in Table 1. Note that the team should perform the activities collaboratively; the idea of *roles* is to have people in the team to lead a specific part of the work and act as a spokesperson for that specific area.

Table 1: Selected roles for the software development team.

Role	Key responsibilities
Team manager (coordinator)	Actively promote the best collaboration in the team and take the initiative to address problems that may arise. Ensure the necessary discussion so there is a fair distribution of tasks and that members work according to the plan. Ensure that the requested project outcomes are delivered in time.
Product owner	Represents the interests of the stakeholders. Has a deep understand of the product and the application domain; the team will turn into the Product Owner to clarify questions about product features/requirements. Responsible for accepting the solution increments.
Architect	Deep understanding of the proposed architecture and supporting technologies. The team will turn into the Architect to explain the expected behavior of each software component and interactions between modules.
DevOps master	Responsible for the infrastructure and its configuration; ensures that the development framework works properly. In-charge of preparing the deployment machine(s)/containers, Git repository, cloud infrastructure, databases operations, etc.
Developer	ALL members contribute to the development tasks.

1.3 Product concept

Each group is expected to propose, conceptualize, and implement a multi-layer, enterprise-class solution observing the guidelines in the following sections.

1.3.1 Generic product requirements

The scope of the proposed solution should include:

- distributed (remote) **generation of data streams**, either from real data sources (e.g.: collecting data from devices) or by virtual software "agents" (e.g.: simulating the behavior of a Stock market)
- **data publishing** using a lightweight, message-oriented protocol (push data upward: remote → central backend).
- long term **storage** of data using a persistent database engine.
- **central processing**, ensuring the detection of relevant events (e.g.: alarms) and data aggregation.
- **service API** (REST) providing a comprehensive set of endpoints to access data and manage the system.

— a **web portal** to implement the core user stories. Using the web portal, the users will be able to track the current updates (access the upstream data) and query/filter stored data (historical views). In addition, you can provide a **mobile application** as an alternative presentation frontend.

The teams should find a suitable application area and validate the scope with the teachers. Application domains such as precision agriculture, health/fitness diaries, smart spaces/homes, smart city infrastructures management, food-deliveries performance tracking systems, etc., provide good examples.

Here is a worked example of the intended scope:

Sample project theme: a wind farms management system.

— **Concept**: an integrated platform to monitor and control remote wind turbines, in wind farms, distributed by multiple sites.
— **Data acquisition layer**: wind turbines telemetry (rpm, energy generation output,...) is being continuous collected; the environment parameters (wind speed, wind orientation,…) are also tracked.
— **Data publishing**: at each wind farm, there is site gateway, acting as a local area aggregator and sends the telemetry data to the cloud, using a bandwidth-savvy protocol.
— **Processing & bizz logic**: hazards conditions are detected when the telemetry reveals that operating thresholds are compromised. In this case, the alarms are also forwarded to the mobile devices of the people in charge. Peak conditions are detected in an hourly basis (data aggregation) to build historic trends.
— **Integration API**: the computational platform exposes endpoints that allows other authorized applications to programmatically list the park information system (e.g.: farm location, details of each tower,…) and telemetry readings, both current and for past intervals of interest.
— **Web portal**: the web application allows basic tracking of operational conditions (telemetry dashboard of the turbines); browse descriptive information on each tower/device (model, location,...); adjust operational parameters, actuating in the wind farm (e.g.: stop a turbine).

1.3.2 Architectural requirements

The solution should adopt, in general, a multilayer architecture as depicted (Figure 1).

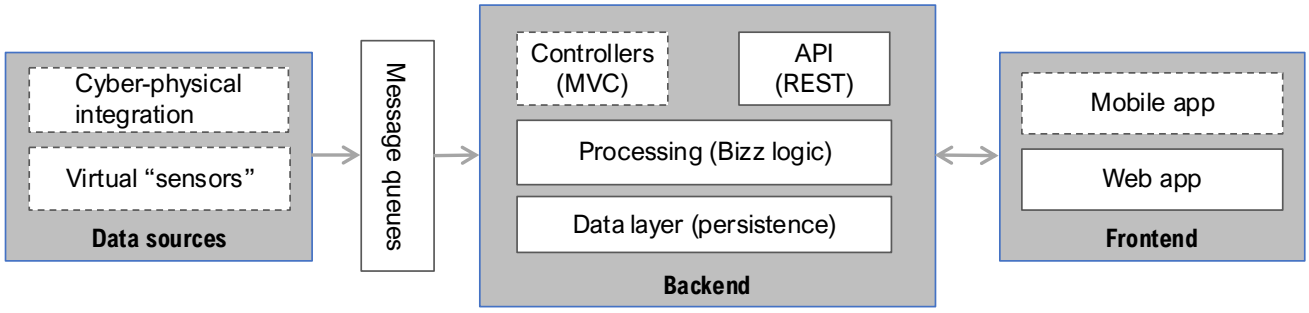

Figure 1: High-level reference architecture (should be adapted and detailed for each project; dashed components may be relevant or not, depending on the approach).

Notes on the expected components and implementation technologies:

a) **Data sources:** the system should ingest data streams, typically generated at cyber-physical systems (e.g.: environmental sensors, cameras, location tracking, devices telemetry, body sensors, etc.) or by virtual software "sensors". The use of hardware devices is optional; you may use "digital twins" instead (virtual representation of a device, simulated in software); or event "agents" that generate data streams that are not related to hardware (e.g.: outcome lab tests for COVID-19).
b) **Message queues**. Data ingestion (from streams into the backend) should use a lightweight message-oriented middleware.
c) **API (REST)**. Endpoints to allow presentation layers (or other systems) to connect to your own services. The API should follow the REST style, with JSON payloads. Note that a comprehensive API would allow you to access (almost) all system services through the API.

d) **Processing & bizz logic**. This module will likely be divided into several sub-modules, depending on the specific problem domain. You should have some processing module to analyze the incoming streams (likely to integrate with the message broker).
e) **Data access**. Data from streams, and the information system to support other parts of the system (e.g.: users, profiles, preferences), is to be persisted using an object-database mapping framework, like Spring Data. The system will save the data into a convenient database technology. While the Relational databases are obvious options (e.g.: Postgresql, MySql), you may use other solutions that better fit the project requirements (e.g.: time-based queries will benefit from a timeseries repository).
f) **Web app**: you may choose the web development framework and integrate with the backend using the services API. If you choose to use a Java-related technology (e.g.: Thymeleaf and Spring MVC), the web layer may use the controllers in the backend (bypassing the API). If you choose a JavaScript-based framework, for example, the presentation layer would interact with the API.
g) **Mobile app** (optional): a simple application that interacts with the backend using the REST API. Push notifications would be an additional challenge.

Additional notes:

h) You need to implement at least one presentation platform: web app or mobile app.
i) There could be other elements, depending on the problem. E.g.: you are not asked to implement actuation (send control instructions from the backend to the devices) but it may be important for your scenario.

1.3.3 Extra credits (advanced topics)

The following challenges are not mandatory, but may give you extra credits in the project, depending on the quality of the implementation:

- Robust data analytics (integrate a framework for streams processing at the backend)
- Instrumentation to monitor the production environment (e.g.: *ELK stack* for application logs analysis)
- Use of relevant cyber physical components (e.g.: sensors or actuators connected to a RPi board) to deliver value to the end-users.

2 Project implementation

2.1 Required practices

2.1.1 Agile *backlog* management (plan & track)

The team will use a backlog to prioritize, assign and track the development work.

This backlog follows the principles of "agile methods" and use the concept of "user story" as the unit for planning. Stories are briefly documented declaring the benefit that a given *persona* wants to get from the system.

Stories have points, which "quantifies" the shared expectation about the effort the team plans for the story, and prioritized, at least, for the current iteration. Developers start work on the stories on the top of the current iteration queue, adopting an agreed workflow.

There are sever options for the backlog management:

— Atlassian Jira (with Scrum boards);
— GitLab Project management (with boards);
— GitHub Projects + Scrum boards;
— PivotalTracker (public projects can use all features).

The backlog should be consistent with the development activities of the team; both the project management environment and Git repository activity log should provide faithful evidence of the teamwork.

For more information on user stories, check this FAQ on story-oriented development (note: you don't need to implement the Acceptance Criteria parts).

2.1.2 Feature-branching workflow

There are several strategies to manage the shared code repository and you are required to adopt one in your team. Consider using the "GitHub Flow"/feature-driven workflow or the "Gitflow workflow".

The **feature-branches should be traceable to user-stories** in the backlog.

Complement this practice by issuing a "pull request" (a.k.a. merge request) strategy to review, discuss and optionally integrate increments in the *main*. All major Git cloud-platforms support the pull-request workflow (e.g.: GitHub, GitLab, Bitbucket).

2.1.3 Containers-based deployment

Your logical architecture should apply the principle of responsibilities segregation. Accordingly, the deployment of services should separate the services into specialized containers (e.g.: Docker containers). Your containers will likely map the architecture logic layers. Your solution needs to be deployed into a server environment (e.g.: Cloud infrastructures) using more than one "slice"/container[1].

2.2 Project outcomes/artifacts

2.2.1 Project repository (Git)

The project outcomes should be organized in a cloud-based Git repository. Besides the code itself, teams are expected to include other project outcomes, such as requested documentation. The project must be shared with the faculty staff. Expected repository structure:

readme.md
🗁 reports/
🗁 presentations/
🗁 *projX*/
🗁 *projY*/

Part	**Content:**
README.md	Context and project bookmark placeholder. Be sure to include the following sections: — Project abstract: title and brief description of the project features. — Project team: students' identification and the assigned roles. — Architecture Diagram: updated diagram each iteration — [Project] Bookmarks: links to quickly access all project resources, such as project management boards, editable versions of the reports in the cloud, entry point for your API documentation,....
🗁 reports/	Project specifications, as PDF files.
🗁 presentations/	Materials used in project-related presentations.
🗁 *projX*/…	The source code for subproject "projX". The amount and names of subprojects depend on each problem.

[1] Although it is likely that your use Docker containers, other options may be considered, depending on the hosting infrastructure.

2.2.2 Project requirements and technical specifications

The project documentation should be kept in the main branch of the repository ([main]/reports) and updated accordingly.

The "Project Specification Report" [see sample template] should cover:

A. Product concept
   A.1. Vision statement
   A.2. Personas
   A.3. Supported scenarios (user stories)
B. Architecture notebook
   B.1. Key quality requirements
   B.2. Architectural view
   B.3. Module interactions [dynamic view]
C. Information model

2.2.3 API documentation

Provide an autonomous report (or a web page) describing the services API. This should explain the overall organization, available methods, and the expected usage. See related example.

Consider using the OpenAPI/Swagger framework to create the API documentation.

2.3 Project schedule: iteration plan

The project will be developed in 2-week iterations. Active management of the product backlog will function as the main source of progress tracking and work assignment.

Expected results from project iterations:

Iter. #	Focus	Required outcomes
I1.1 16/10	Project initiation (define the concept, setup the tools, prioritize user stories).	• Core stories defined and prioritized. → Project Specification (report), part A. • Backlog management system setup. • Team repository in place.
I1.2 06/11	Define the product architecture. UX preview.	• Draft Project Specification (report); must include the Architecture Notebook part. • Prototypes[2] for the core user stories.
I2 13/11 20/11	Develop a few core user stories involving **data access**. Demonstrate the architecture end-to-end (full-stack proof-of-concept: from data generation to user-interface).	• Basic data pipeline in-place: data streams generation, transmission, and storage. • Product increment covering (at least) a user story related to data access (browse current values from streams, in the web presentation layer.) • Increment deployed (in containers) at the server environment.

[2] These "prototypes" would be early versions of the web pages, essentially "static" (not yet integrated with the business logic), already implemented with the target technologies. If not possible, then mockups).

Iter. #	Focus	Required outcomes
I3 27/11 04/12	Develop a few user stories requiring data processing. Stabilize the API and system integrations.	• Additional user stories required for a functional MVP deployed, specially covering data aggregation/ visualization of historic data. • Required user story: alarms/events detection on data streams and feedback to the UI. • REST API deployed in the server. • Implement integrations with external services (if applicable, e.g.: consuming public web services) • Integrate the cypher-physical layer (if applicable)
I4 11/12 18/12	Stabilize the Minimal Viable Product (MVP). Present the first release of the MVP.	• Stabilize the presentation layer (for end-users) • Stabilize the REST API. • Stabilize the production environment. • MVP backend deployed in the server (or cloud); relevant/representative data included in the repositories (not a “clean state”). • Update documentation (project specifications and software documentation). • Oral presentation/defense.