TQS: Project Assignment Guidelines

I. Oliveira, v2022-05-09

1 Project objectives and iterations

1.1 Project assignment objectives

Students should work in teams to specify and implement a medium-sized project, comprising:

- Development of a viable software product (functional specification, system architecture and implementation).
- Specification and enforcement of a *Software Quality Assurance* (SQA) strategy, applied throughout the software engineering process.

1.2 Team and roles

While the team may assign specific roles to different members, all students need to contribute as *developers* to the solution. Each team/group should assign the following roles:

Role	Responsibilities
Team Coordinator (a.k.a. *Team Leader*)	Ensure that there is a fair distribution of tasks and that members work according to the plan. Actively promote the best collaboration in the team and take the initiative to address problems that may arise. 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 to the Product Owner to clarify the questions about expected product features. Should be involved in accepting the solution increments.
QA Engineer	Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure que quality of the deployment. Monitors that team follows agreed QA practices.
DevOps master	Responsible for the development and production infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparing the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.
Developer	ALL members contribute to the development tasks.

1.3 Iterations plan

The project will be developed in 1-week iterations. Active management of the product backlog will act as the main source of progress tracking and work assignment.
Each "Prática" class will be used to monitor the progress of each iteration; column on the right lists the minimal outcomes that you should prepare to show in class.

Expected results from project iterations:

Iter. # (start)	**Outcomes to be presented in class (*Práticas*)**	**Main focus/activities for the iteration**
I0, prep 12/5→	n/a [project quick-off]	• Define the product concept. • Outline the architecture. • Team resources setup: code repository in place, shared documents, chat channel,…
I1 19/5	• Present the product concept. • Software/system architecture proposal.	• Define the product architecture. • Outline the SQE tools and practices. • Product specification report (draft version; product concept & requirements section) • Backlog management system setup. • UI prototyping (partial)
I2 26/5	• Present your SQE strategy. • UI prototypes[1] for the core user stories. • Plan for next iteration: user stories to address and implement.	• A few core stories detailed and implemented. • QA Manual (draft version) • CI Pipeline.
I3 02/06	• Product increment with (at least) a couple of core user stories: search + buy/order. • CI Pipeline and merge-requests policy. • Plan for next iteration: user stories to address and implement.	• Develop a few core user stories involving data access/persistence. • Comprehensive API. • Set up the CD pipeline. • QA Manual (final version)

[1] These "prototypes" would be early versions of the web pages, already implemented with the target technologies (and not mockups) but more or less "static" (not yet integrated with the backend/business logic).

Iter. # (start)	**Outcomes to be presented in class (*Práticas*)**	**Main focus/activities for the iteration**
I4 09/6	• Comprehensive REST API. • CD Pipeline: services deployed to containers (or cloud). • Quality dashboard and gates.	• Stabilize the Minimal Viable Product (MVP). • All deployments available in the server. • relevant/representative data included in the repositories (not a "clean state"). • Product specification report (final version.).
17/6 or 23/6	• Minimal Viable Product (MVP) backend deployed in the server (or cloud). Oral presentation/defense.	P2+P3 → present on 17/6. P1+P4 → present on 23/6.

2 Product definition

2.1 General requirements of the project

To make the projects comparable across teams, with respect to scope and difficulty, there is a general definition of terms for what the project should include.

You project should implement a digital marketplace for "last-mile" deliveries, similar to initiatives like Glovo, UberEats, Roadie,…
All the e-commerce sites in this area share some common needs: manage a dynamic workforce of riders; accept orders; optimize the assignment of jobs to riders; manage riders reputation; etc. In your project, you should clearly separate what is common to several businesses (deliveries/logistics services); and what is the specific area of the store (that uses the deliveries platform).

This includes two main "sub-projects":

a) The **deliveries platform** ("engine"), with use cases such as riders' registration and reputation management, dynamic matchmaking of orders and riders, performance dashboard, etc.
b) A **specific application** /market proposition that leverages on the deliveries platform, e.g., food ordering, drugs (medicines) delivery, "small" marketplace promoted by a municipality to stimulate local stores, etc.

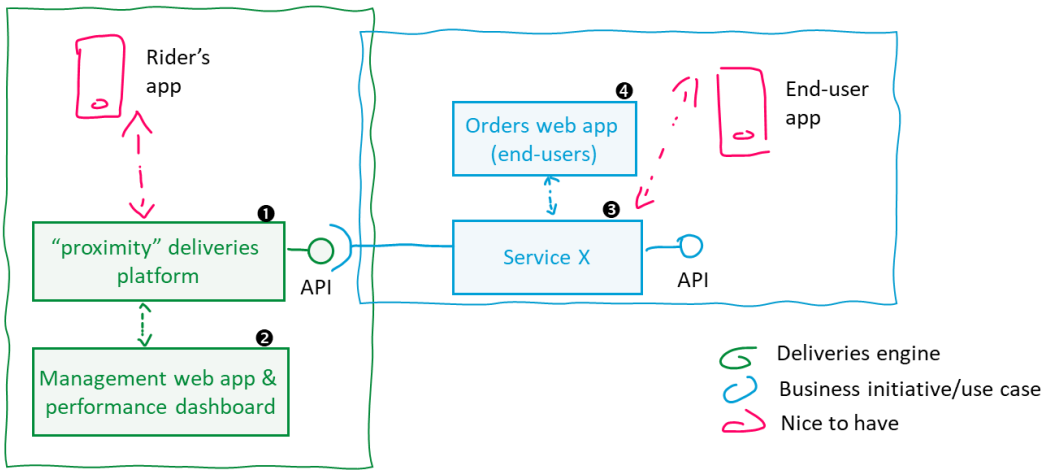

Figure 1

The "deliveries engine" should (by design, at least) be reusable in different contexts/applications, i.e., it captures the usual requirements of the last-mile deliveries, with dynamic matchmaking of rides and riders, and offers an

implementation that could be used by different end-user applications[2]. This "engine" provides the technological platform for aggregation/intermediation. Businesses use the platform to submit, track and assess their delivery requests.
You should also create a simple demonstrator for a possible (end-user facing) application/business that integrates with the delivery's platform.
Here is an example of a hypothetic scenario:

a) The "Snow-White-in-Rush" (SWiR) is a small business that offers a laundry service. Digital channels are available for their customers to place orders, including a mobile app and a web app.
b) SWiR is launching a new premium service, for urgent cleaning orders, in which the laundry is picked at the customer's home by a rider/courier.
c) To quickly deploy this new service, SWiR is establishing a partnership with the CityConnect platform which offers a convenient solution for "local deliveries as a service".
d) The comprehensive CityConnect API provides all the services needed for the SWiR small business to place and monitor delivery requests.

Teams should develop requirements/use cases for their own case study and propose a solution considering the architectural constraints defined in section 2.2.

2.2 Implementation constraints

The following components should appear in the overall architecture of the solution (other modules can added to the solution):

- **Backend** implemented with Spring Boot/Java EE technology. The *backend* includes the business logic, persistence (database) and at least one REST API (can provide additional services, e.g. *streaming)*. (see ❶,❸ in Figure 1)
- **Web app(s)** that corresponds to the main interface of the marketplace, online. (see ❷, ❹ in Figure 1)
- **Nice to have:** a simple Mobile application intended for the rider (or final consumer).

3 QA and development practices

3.1 Practices to implement

3.1.1 Agile (user-stories) *backlog* management

The team will use a backlog to prioritize, assign and track the development work. This backlog follows the principles of "agile methods" and should use the concept of "user story" as the unit for planning.
Stories have points, which reveal 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.
Stories have acceptance criteria. This will provide the context/examples to write user-facing tests (or inspire other tests).
The backlog should be consistent with the development activities of the team; both the backlog and Git repository activity log should provide a faithful evidence of the teamwork.
For more information on user stories, check this FAQ on story-oriented development (note: Acceptance Criteria **is required**).

[2] As an example of the reusable "platform" concept, see, for example, the product https://gamifyou.com/ ; it provides the "engine", through comprehensive API, but you still need to develop the specific use case.

3.1.2 Feature-branching workflow with code reviews

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-branch should match the user-story** in the backlog.
You are expected to complement this practice by issuing a "pull request" (a.k.a. merge request) to review, discuss and optionally integrate increments in the *master*. All major Git cloud-platforms support the **pull-request abstraction** (e.g.: GitHub, GitLab, Bitbucket). Static code analysis should be performed before accepting the pull-request.

3.1.3 Software tests

Software developers are expected to deliver both production code and testing code. The "definition of done", establishing the required conditions to accept an increment from a contributor, should define what kind of tests are required.
There should be traceability between the user story acceptance criteria and the tests included in an increment. Projects should provide evidence if they use **TDD practices** (which are recommended).

3.1.4 CI/CD pipeline

The team should define, setup and a adopt a CI/CD pipeline. The project should offer evidence that the contributions by each member go through a CI pipeline, explicit including tests and quality gates assessment (static code analysis). The team should also offer CD, either on virtual machines or in cloud infrastructures.

3.1.5 Containers-based deployment

Your logical architecture should apply the principle of responsibility segregation to identify layers. Accordingly, the deployment of services should separate the services into specialized containers (e.g.: Docker containers). Your solution needs to be deployed to a server environment (e.g.: Cloud infrastructures) using more than one "slice".

3.2 Project outcomes/artifacts

3.2.1 Project repository

A cloud-based **Git repository** for the project code and reporting. The main submission channel will be the 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 structure:

```
README.md
docs/
projX/
projY/
```

...with the following content:

- docs/ → reports should be included in this folder, as PDF files, specially the QA Manual and the Project Specification report.
- projX/ → the source code for subproject "projX", etc.
- **README.md** → be sure to include an informative README with the sections:
  a) Project abstract: title and concise description of the project.
  b) Project team: students' identification (and the assigned roles, if applicable).
  c) Project bookmarks: link **all relevant resources** here!
     - Project Backlog (e.g.: JIRA, GitLab agile planning)
     - Related repositories (there may be other related code repositories...)

- Related workspaces in a cloud drives, if applicable (e.g.: shared Google Drive)
- API documentation (link the landing page for your API documentation)
- Static analysis (quick access to the quality dashboard)
- CI/CD environment (quick access to the results dashboard)
- ...

For most CI/CD pipelines, it would be **better to use more than one repo**, i.e., specific git repositories for different projects/modules. In that case, consider:

- using a “main repository” and be sure to link related repositories in the README.md, or
- create a “group” of repositories (if your cloud infrastructure has that concept) and share the group.

3.2.2 Project reporting and technical specifications

The project documentation should be kept in the master branch of the repository (folder: /docs) and updated accordingly.
There are two main reports to be (incrementally) prepared:

1. Product Specification report [→ template available in the project resources]
2. QA Manual [→ template available in the project resources]

3.2.3 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, Postman documentation, or similar, to create the API documentation.

3.3 Extra credit

The following challenges are not mandatory, but will give you extra credits in the project:

- Instrumentation to monitor the production environment (e.g.: Nagios alarms, *ELK stack* for application logs analysis,...)
- Collaborate with other group to integrate their deliveries engine in your business use case (B2B scenario). You still need to implement your own deliveries engine, but, with this integrating, you would be able to work with more than one delivery engine.