



Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio - Ingeniería de Software Ágil 2

[Link al repositorio](#)

Entrega 5

Integrantes:

Danilo Biladoniga - 231749

Tomás Núñez - 257564

Germán Oller - 242312

Índice

Resumen de gestión del proyecto	2
Reflexiones sobre el aprendizaje	4
Aplicar un marco de gestión ágil	4
Analizar deuda técnica	4
Implementar un repositorio y procedimientos de versionado	4
Crear un pipeline con eventos y acciones	5
Integrar prácticas de QA en el pipeline y gestionar el feedback	5
Generar escenarios de testing desde la perspectiva del usuario	6
Automatizar el testing funcional o de caja negra	6
Reflexionar sobre DevOps	6
Lecciones aprendidas	7
Configuración de adecuada del archivo .gitignore:	7
Uso de TDD:	7
Configuración de CORS:	7
Conclusiones	9
Guía de instalación para desarrollo y despliegue en producción	10
Instalación de herramientas y programas necesarios	10
Restauración de la base de datos	11
Ejecución de pruebas	11
API	11
Integración con el Cliente	13
Ejecutar API y Cliente	14
Ambiente de Desarrollo	14
Ambiente de Producción	15
API	15
Cliente Angular	16

Resumen de gestión del proyecto

El proyecto se centra en la implementación de los principios de DevOps con el propósito de mejorar una aplicación asignada. Para alcanzar este objetivo y mantener una alta productividad, se han aplicado prácticas clave, tales como la integración continua, la entrega continua, pruebas automáticas y control de calidad (QA).

En la gestión del proyecto, optamos por emplear Kanban como marco ágil. Fundamentamos esta elección basándonos en los requisitos específicos del proyecto. Dentro de este marco, se llevaron a cabo cuatro entregas, con intervalos que oscilaron entre una y tres semanas. Al concluir cada entrega, el equipo se reunía en retrospectivas para evaluar el cumplimiento de los objetivos y reflexionar sobre el proceso en curso.

Además para la medición de la productividad del equipo se usaron diversas métricas como Lead Time, Cycle Time, Flow Efficiency y Throughput.

Sobre el Lead Time, los valores obtenidos fueron:

- Promedio para Bug fixes = 4 días
- Promedio para nuevas features: 20 días

La notoria diferencia entre estos Lead Times se debe a la tecnología asociada a las diferentes tareas.

Este también es el motivo de las diferencias en el Cycle Time:

- Promedio para Bug fixes = 1:30 horas
- Promedio para nuevas features: 5 días

Estos fueron los resultados obtenidos de Flow Efficiency:

- Para Bug fixes = 1.5625
- Para nuevas features = 1.0416

Por último, el Throughput por cada entrega fue:

Entrega 1:

Throughput para los bugs: 0

Throughput para las funcionalidades: 0

Entrega 2:

Throughput para los bugs: 3

Throughput para las funcionalidades: 0

Entrega 3:

Throughput para los bugs: 0

Throughput para las funcionalidades: 3

Entrega 4:

Throughput para los bugs: 0

Throughput para las funcionalidades: 0

Para más detalles sobre métricas en el proyecto, ver documento “Entrega 4” (dentro de Documentacion/Entregas ISA2/Entrega 4 en el repositorio de GitHub).

Para la implementación de nuevas funcionalidades, se optó por la metodología BDD (Behavior Driven Development o Desarrollo Guiado por Comportamiento). Una vez completadas las nuevas funcionalidades, se llevaron a cabo revisiones del producto con el Product Owner designado, Tomás Núñez. Estas revisiones permitieron asegurar la alineación de las implementaciones con las expectativas y requisitos del cliente, facilitando así una integración efectiva de las nuevas funcionalidades en el producto final.

Reflexiones sobre el aprendizaje

Aplicar un marco de gestión ágil

Cómo ya mencionamos, utilizamos Kanban como marco de gestión. Sin embargo, no lo hicimos de manera “tradicional”, ya que consideramos que sería mejor para el proyecto adaptar algunas prácticas y principios a nuestras necesidades. Fuimos capaces de realizar estas adaptaciones gracias a que mantuvimos una muy buena comunicación a lo largo de todo el proyecto, aspecto que deberíamos mantener en futuros trabajos.

Aprendimos a ser más flexibles, colaborativos y receptivos al cambio, lo que nos permitió mejorar nuestra gestión del proyecto y alcanzar mejores resultados en cada entrega.

Analizar deuda técnica

Para analizar la deuda técnica, realizamos pruebas exploratorias, pruebas de usabilidad y análisis del código fuente. Antes de comenzar, intentamos comprender el negocio y sus requisitos, aunque quizás no invertimos el tiempo necesario para comprenderlo completamente.

Algo que deberíamos cambiar de cara al futuro es el apresurarnos a comenzar con el análisis. Una investigación más en profundidad de las herramientas de análisis de código nos habría facilitado el trabajo realizado.

Un aspecto positivo que podemos destacar, y deberíamos repetir en el futuro, es la planificación de las pruebas, ya que fuimos capaces de encontrar una cantidad significativa de errores.

Implementar un repositorio y procedimientos de versionado

Consideramos haber realizado una buena implementación del repositorio y una correcta aplicación de los procedimientos de versionado.

La decisión de definir claramente una estructura de carpetas, así como una nomenclatura de ramas, fue clave para los resultados obtenidos. Establecer estos criterios de manera temprana en el proyecto (en la Entrega 1) nos permitió acostumbrarnos a la forma de trabajo. Definitivamente deberíamos mantener estas prácticas de cara al futuro.

Un problema al que nos enfrentamos fue a la configuración del archivo `.gitignore`. Al no tenerlo bien configurado, tuvimos que solucionar que no se subieran archivos innecesarios al repositorio.

Crear un pipeline con eventos y acciones

Fuimos capaces de crear un pipeline robusto, que fue clave en el proyecto. Una práctica realizada, que nos resultó muy útil y sin dudas deberíamos hacer siempre que estemos frente a una tecnología nueva (cómo era el caso del pipeline en GitHub Actions) es un estudio previo a la implementación. Nos basamos en documentación oficial y en ejemplos ya hechos, lo cuál nos permitió realizar esta implementación sin demasiados problemas. Quizás algo a mejorar es la ambición; en nuestro caso cuando terminamos el pipeline buscamos mejorarlo, e invertimos tiempo en un aspecto que finalmente no pudimos poner en práctica.

Integrar prácticas de QA en el pipeline y gestionar el feedback

Pudimos aprovechar las ventajas de agregar pruebas unitarias y utilizar el pipeline de GitHub Actions. Las pruebas unitarias nos ayudaron a evaluar la funcionalidad y la integridad del nuevo código, garantizando estándares de calidad. El uso de GitHub Actions automatizó el proceso de ejecución de pruebas y proporcionó un feedback rápido, lo que nos permitió tomar medidas proactivas para solucionar los problemas identificados. Esta experiencia nos reafirmó la importancia de integrar prácticas de QA en el desarrollo y utilizar herramientas que aseguren la calidad del software de manera rápida y eficiente.

Generar escenarios de testing desde la perspectiva del usuario

Utilizamos BDD y Speck Flow como herramientas para abordar este enfoque. Sin embargo, reconocemos que podríamos haber realizado más pruebas y sobre todo agregar una mayor cantidad de casos.

Un estudio más detallado del tema, así como más práctica con las herramientas, nos hubiesen facilitado mucho esta tarea.

Automatizar el testing funcional o de caja negra

Al utilizar Selenium para automatizar el testing funcional, pudimos explorar y evaluar el frontend de la aplicación de manera más exhaustiva. Sin embargo, nos enfrentamos al desafío de asegurar la disponibilidad y consistencia de los datos en la base de datos de pruebas.

Aprendimos la importancia de planificar y gestionar adecuadamente los datos necesarios para los casos de prueba, estableciendo procesos sólidos para garantizar una evaluación confiable y eficiente.

Nos encontramos con algunos problemas de Cors y Selenium, ya que no habíamos agregado la flag `acceptInsecureCerts=true`, que se agrega para permitir acceder a direcciones http de la API. Tuvimos que invertir tiempo para solucionar este problema, algo que ya aprendimos y podemos solucionar rápidamente en el futuro.

Reflexionar sobre DevOps

Las técnicas de DevOps tuvieron un impacto significativo en la calidad y productividad del equipo. En comparación a cómo trabajamos antes de aplicar las técnicas de DevOps (en otros obligatorios) observamos mejoras en la calidad del software, una mayor eficiencia en el desarrollo y en mejoras en cada una de las entregas, así como una colaboración más estrecha entre los integrantes del equipo. Sin embargo, también enfrentamos desafíos, como la necesidad de adquirir nuevas habilidades. A pesar de las dificultades, consideramos que los beneficios superan ampliamente los obstáculos y estamos comprometidos a seguir mejorando en esta área clave de desarrollo de software.

Lecciones aprendidas

Configuración de adecuada del archivo .gitignore:

La configuración adecuada del archivo .gitignore es esencial para evitar conflictos innecesarios en archivos dentro de las soluciones de .NET. Estos conflictos, que surgen al no ignorar correctamente ciertos archivos o directorios, pueden ser difíciles de depurar y resolver, especialmente cuando van más allá de nuestro conocimiento inmediato.

La importancia de mantener un repositorio limpio de archivos innecesarios radica en la prevención de conflictos no deseados que podrían ralentizar el progreso del proyecto. La conclusión extraída de esta experiencia resalta la necesidad crítica de una gestión cuidadosa de la configuración en el archivo .gitignore para evitar complicaciones que puedan surgir debido a la inclusión inadvertida de archivos temporales, compilados o dependencias no esenciales. Una práctica sólida en este sentido contribuye significativamente a la estabilidad y fluidez del desarrollo del proyecto, al tiempo que minimiza los desafíos asociados con la resolución de conflictos.

Uso de TDD:

Lamentamos no haber cumplido al 100% con TDD (Desarrollo Guiado por Pruebas). Esta omisión podría haber causado problemas potenciales, como la introducción de errores y una posible baja en la calidad del código. Aunque hasta ahora no hemos observado una disminución evidente en la calidad, reconocemos la importancia de seguir las mejores prácticas de desarrollo para prevenir futuros desafíos.

Configuración de CORS:

Hemos aplicado la configuración CORS, especialmente al trabajar con Selenium. Aprendimos que al incluir la flag `acceptInsecureCerts=true`, permitimos el acceso a direcciones HTTP de la API, facilitando la integración y la interacción efectiva con Selenium. Esta configuración nos ha permitido aprovechar al máximo las

capacidades de Selenium en entornos que requieren comunicación con API a través de protocolos HTTP.

Conclusiones

Luego de haber realizado todas las entregas podemos concluir que la gestión de un proyecto es igual o hasta más importante que el desarrollo en el proyecto mismo, ya que organiza a cada integrante del equipo y mantiene un flujo de trabajo continuo y ordenado.

Como fue hablado en el curso y vivido en experiencia propia, existe un concepto de experimentación continua y aprendizaje, el cual refiere a manejar, aprender y dominar herramientas que faciliten y ayuden en el proceso.

Un claro ejemplo fue la automatización de algunas tareas como los test y el compilado antes de cerrar una pull request, implementada por Git Actions. Con estas tareas automatizadas el equipo puede enfocarse en agregarle valor al producto y evitar, en gran parte, el retrabajo de funciones previamente desarrolladas. También se logra reducir costos (consecuencia directa de disminuir el retrabajo), reducir riesgos y ahorrar tiempo.

Estas herramientas fomentan el Continuous Deployment, brindándole siempre valor al producto entregado. A su vez, más puestas en producción llevan a más aprendizaje del equipo, el cual trasladado al resto de equipos genera una cultura de aprendizaje de la que todos se ven beneficiados.

Además produce que el equipo se vuelva más eficiente, brindando mayor calidad al software, y hace los deploys más frecuentes, volviendo este aspecto clave del proceso de ingeniería de software parte del día a día.

Kanban nos guió en este proceso, ofreciendo un flujo de entrega continua. En el contexto de desarrollo de nuevas funcionalidades o productos, el uso de Sprints u iteraciones de Scrum es de suma ayuda. En un contexto de mantenimiento optaríamos por utilizar un tablero de Kanban, el cual favorece el Continuous Deployment, aportando valor en el menor tiempo posible al cliente.

Guía de instalación para desarrollo y despliegue en producción

Instalación de herramientas y programas necesarios

Asegúrese de tener instalados los siguientes elementos para el correcto funcionamiento e instalación de las aplicaciones.

[.NET 6.0 SDK](#)

Plataforma de desarrollo que permite ejecutar la API y sus correspondientes tests unitarios.

[Entity Framework](#)

Utilizado para generar y aplicar migraciones de la base de datos.

[SQL Server Management Studio \(SSMS\)](#)

Gestor de bases de datos SQL.

[Node.js](#)

Runtime para entornos javascript, el cual se utiliza para ejecutar la aplicación cliente.

[Angular CLI](#)

Utilizado para desarrollar, mantener y ejecutar la aplicación cliente de Angular.

[Selenium IDE](#), [Command-line Runner](#) y drivers de acuerdo a su navegador
([chromedriver](#), [geckodriver](#) y/o [edgedriver](#))

Utilizado para ejecutar pruebas automatizadas.

Restauración de la base de datos

En primer lugar vamos a proceder con la restauración de la base de datos, el cual el archivo que vamos a utilizar se encuentra en el repositorio, más en específico en la siguiente ruta “Codigo\Backups\BaseDeDatosConDatos\PharmaGoDb.bak”.

Para realizar esto es necesario ejecutar la herramienta SQL Server Management Studio, nos dirigimos a la carpeta “Databases” y al presionar botón derecho, “Restore database...” se nos abrirá una ventana.

Una vez en dicha ventana seleccionaremos la opción de “Device” y en la parte derecha elegimos el archivo previamente mencionado, esperamos a que termine, quedando de esta forma la restauración completada y podremos continuar con las siguientes partes.

Recordatorio: en caso de nombrar la base de datos de una forma distinta o tener configuraciones específicas, recuerde modificar el ConnectionString que se encuentra en la siguiente ruta:

“Codigo\Backend\PharmaGo.WebApi\appsettings.json”

Ejecución de pruebas

En esta parte vamos a ejecutar las pruebas mediante comandos en consola. Las hemos dividido en dos pasos para detallar y facilitar la ejecución de cada una de las mismas.

API

Antes de continuar debemos instalar las dependencias del proyecto de la API, lo que se realizará ejecutando el siguiente comando desde la raíz de repositorio:

```
cd .\Codigo\Backend\ | dotnet restore
```

Para la ejecución de pruebas unitarias hechas con MSTest y las correspondientes pruebas hechas en SpecFlow realizaremos lo siguiente:

En primera instancia debemos ejecutar la API para poder correr las pruebas de SpecFlow correctamente.

Para esto abrimos una terminal y desde la raíz del repositorio ejecutaremos el siguiente comando:

```
dotnet run --project .\Codigo\Backend\PharmaGo.WebApi\
```

Alternativa

Parados en la carpeta “\Codigo\Backend\PharmaGo.WebApi” ejecutar:

```
dotnet run
```

Una vez ejecutada la API correctamente, abriremos una nueva terminal y desde la raíz del repositorio ejecutaremos la siguiente sentencia:

```
dotnet test .\Codigo\Backend\
```

Alternativa

Parados en la carpeta “\Codigo\Backend\” ejecutar:

```
dotnet test
```

Al cabo de unos segundos/minutos se ejecutarán todas las pruebas en dicha terminal dándonos una salida similar a la siguiente imagen:

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed:    0, Passed:    7, Skipped:    0, Total:    7, Duration: 180 ms - SpecFlowPharmaGo.WebApi.dll (net6.0)
PharmaGo.Test -> D:\ORT\8\Agil2\obligatorio-biladoniga-nunez-oller\Codigo\Backend\PharmaGo.Test\bin\Debug\net6.0\PharmaGo.Test.dll
Test run for D:\ORT\8\Agil2\obligatorio-biladoniga-nunez-oller\Codigo\Backend\PharmaGo.Test\bin\Debug\net6.0\PharmaGo.Test.dll (.NETCoreApp,Version=v6.0)
Microsoft (R) Test Execution Command Line Tool Version 17.3.3 (x64)
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed:    0, Passed:   309, Skipped:    0, Total:   309, Duration: 1 s - PharmaGo.Test.dll (net6.0)
```

Integración con el Cliente

Para esta parte ejecutaremos las pruebas realizadas usando Selenium. En nuestro caso, las mismas serán ejecutadas en chrome. Antes de continuar debemos restaurar la base de datos como se menciona en secciones anteriores.

En primer lugar, necesitamos ejecutar la API y la aplicación cliente en Angular. Para lo primero basta con ejecutar lo mencionado en la parte anterior. Respecto a la aplicación en Angular, realizaremos lo siguiente:

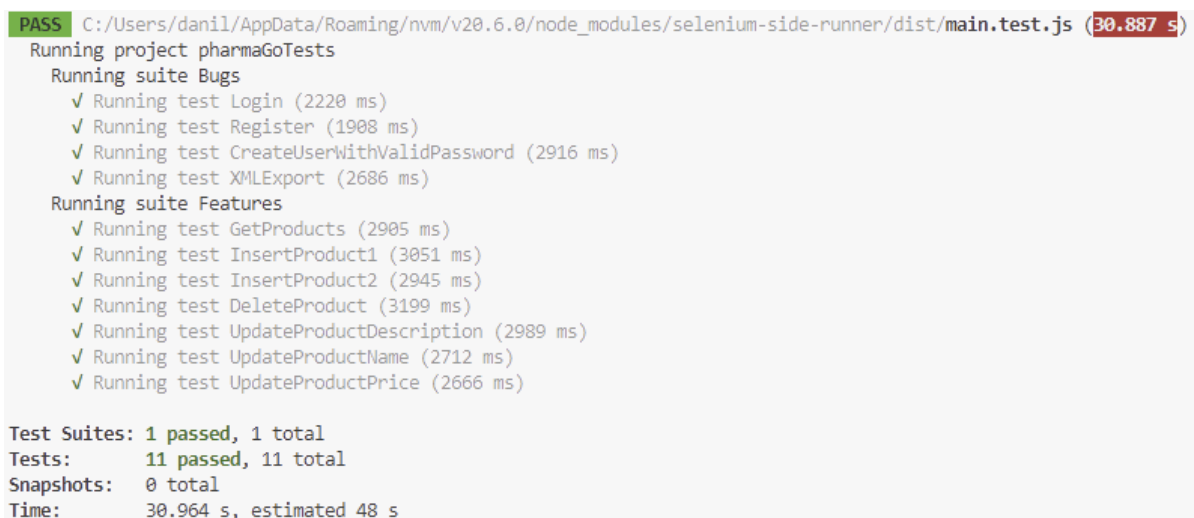
Parados en la raíz del repositorio, ejecutar en una nueva terminal. Esto nos moverá al directorio de la aplicación cliente. Posteriormente, ejecutar en la misma:

```
cd .\Codigo\Frontend\ | npm run start
```

Una vez teniendo ejecutada la API y la aplicación cliente abriremos una nueva terminal. Parados en la raíz del repositorio ejecutaremos el siguiente comando para ejecutar las pruebas de Selenium:

```
selenium-side-runner -c "browserName=chrome  
acceptInsecureCerts=true" .\Codigo\pharmaGoTests.side
```

Una vez terminada la ejecución se mostraría un resultado similar a la siguiente imagen:



```
PASS C:/Users/danil/AppData/Roaming/nvm/v20.6.0/node_modules/selenium-side-runner/dist/main.test.js (30.887 s)
Running project pharmaGoTests
Running suite Bugs
  ✓ Running test Login (2220 ms)
  ✓ Running test Register (1908 ms)
  ✓ Running test CreateUserWithValidPassword (2916 ms)
  ✓ Running test XMLExport (2686 ms)
Running suite Features
  ✓ Running test GetProducts (2905 ms)
  ✓ Running test InsertProduct1 (3051 ms)
  ✓ Running test InsertProduct2 (2945 ms)
  ✓ Running test DeleteProduct (3199 ms)
  ✓ Running test UpdateProductDescription (2989 ms)
  ✓ Running test UpdateProductName (2712 ms)
  ✓ Running test UpdateProductPrice (2666 ms)

Test Suites: 1 passed, 1 total
Tests:      11 passed, 11 total
Snapshots:  0 total
Time:       30.964 s, estimated 48 s
```

Ejecutar API y Cliente

Ambiente de Desarrollo

Para ejecutar la API, desde la raíz del repositorio nos moveremos al directorio “Codigo\Backend\PharmaGo.WebApi” y ejecutaremos el siguiente comando:

```
cd .\Codigo\Backend\PharmaGo.WebApi | dotnet run
```

Antes de continuar debemos instalar las dependencias del proyecto de Angular, parados en la raíz del repositorio ejecutaremos:

```
cd .\Codigo\Frontend\ | npm i
```

Por otro lado, para ejecutar la aplicación de Angular desde la raíz del repositorio abriremos una nueva terminal y nos moveremos al directorio “Codigo\Frontend” y ejecutaremos `npm run start`, comando a continuación:

```
cd .\Codigo\Frontend | npm run start
```

En caso de devolver error de CORS recuerde habilitar el dominio `http://localhost:5186` correspondiente a la API. Para esto lo único que se debe hacer es ir a una nueva pestaña del navegador chrome con dicha URI, mostrar opciones avanzadas y proceder al dominio localhost como se muestra a continuación.



Your connection is not private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards). [Learn more](#)

NET::ERR_CERT_AUTHORITY_INVALID



To get Chrome's highest level of security, [turn on enhanced protection](#)

Hide advanced

Back to safety

This server could not prove that it is **localhost**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

[Proceed to localhost \(unsafe\)](#)

Ambiente de Producción

API

Parados en la raíz del repositorio ejecutaremos el siguiente comando para crear la versión de producción correspondiente al proyecto de la API:

```
cd .\Codigo\Backend\PharmaGo.WebApi\ | dotnet build --configuration  
Release
```

Una vez terminada la ejecución de dicho código se creara la carpeta Release dentro de la ruta “Codigo\Backend\PharmaGo.WebApi\bin\Release\net6.0” la cual contiene la aplicación para producción

“Codigo\Backend\PharmaGo.WebApi\bin\Release\net6.0\PharmaGo.WebApi.exe”

Cliente Angular

Parados en la raíz del repositorio ejecutaremos el siguiente comando para crear la versión de producción correspondiente al proyecto de Angular:

```
cd .\Codigo\Frontend\ | npm run build
```

Una vez terminada la ejecución de dicho comando se genera la siguiente carpeta “Codigo\Frontend\dist\pharma-go” la cual contiene la versión para producción.