código facilito

Bootcamp TESTING AUTOMATIZADO

codigofacilito.com/**bootcamps/testing-automatizado**

códigofacilito

Acerca de Código Facilito

Código Facilito es la **plataforma de aprendizaje** profesional hecha por y para **desarrolladores** de habla hispana.

Desde el 2010 brindando contenido educativo de alta calidad de la mano de **expertos de la industria**.

códigofacilito

Objetivos

Formar profesionales capacitados para diseñar, desarrollar e implementar frameworks de automatización de pruebas con Python, enfocados en aplicaciones web y APIs, utilizando herramientas como Selenium, PyTest, Requests y Behave, así como aplicar principios básicos de CI/CD para integración continua.

Objetivos Específicos

- Adquirir habilidades en el uso de Python aplicado a pruebas automatizadas.
- Dominar el diseño y desarrollo de suites automatizadas para pruebas API y UI.
- Aprender y aplicar patrones de diseño como Page Object Model (POM) para optimizar el mantenimiento y la escalabilidad de las pruebas.
- Implementar prácticas y herramientas de Behavior Driven Development (BDD) usando Behave.
- Integrar automatización de pruebas en pipelines de Integración Continua y Entrega Continua (CI/CD).
- Desarrollar un proyecto final integral que refleje la aplicación de los conocimientos adquiridos durante el bootcamp.

códigofacilito

Formato del Bootcamp

12 Semanas

Clases en vivo cada semana. Grabaciones disponibles si te pierdes la clase en directo.

Grupo Exclusivo

Cada Bootcamp tiene su grupo exclusivo con estudiantes y Team Código Facilito.

6 Meses Premium

Al instante de inscribirte recibes 6 meses de acceso ilimitado a todos los cursos.

Proyecto y Certificado

Completa el Bootcamp con un proyecto final y recibe tu certificado de finalización.

código facilito

Líder de Bootcamp:

HÉCTOR VEGA

Software Development Engineer in Test @ Unosquare

TerranigmArk

Es Ingeniero de Automatización de QA, apasionado por el aprendizaje continuo y la colaboración en comunidades, usando herramientas como Playwright, Selenium, Pytest y Postman. Actualmente trabaja como SDET en Unosquare y Tutor de QA en TripleTen LatAm, donde ha enseñado pruebas web, UI, API y automatización a más de 10 cohortes. Ha liderado migraciones y desarrollado pruebas automatizadas en Cruisebound, Inc. y Distillery, y trabajó en Tributi con pruebas automatizadas de UI. Su enfoque está en la mejora de procesos de prueba y la enseñanza en QA.

códigofacilito

Perfil de Ingreso

- Conocimientos básicos en aseguramiento de calidad (QA manual)
- Conocimientos en técnicas de diseño de pruebas como clases de equivalencia y valores límite aplicados a pruebas positivas y pruebas negativas.
- Conocimientos básicos de programación orientada a objetos (POO).
- Experiencia previa en el uso de control de versiones con Git y GitHub.
- Capacidad para resolver problemas técnicos y trabajar de manera autónoma.
- Disposición para trabajar en proyectos colaborativos y prácticos.

códigofacilito

Perfil de egreso

Al concluir el bootcamp, el egresado será capaz de:

- Diseñar y ejecutar pruebas automatizadas utilizando PyTest, Selenium, Requests y Behave.
- Desarrollar frameworks robustos y mantenibles de automatización para pruebas web y APIs.
- Implementar soluciones automatizadas efectivas y escalables con el enfoque Page Object Model.
- Configurar y utilizar herramientas de integración y entrega continua (CI/CD).
- Desarrollar soluciones automatizadas desde la conceptualización hasta la implementación, facilitando la detección temprana de errores y mejorando significativamente la calidad del software.
- Contribuir eficazmente en equipos de desarrollo ágil, aplicando buenas prácticas en automatización de pruebas.

códigofacilito

Plan de estudio

Semana 1: Introducción a la Automatización QA y Entorno de Trabajo

- **Clase 1: Conceptos básicos de QA Automation y preparación del entorno**
  - Introducción a la automatización de pruebas: objetivos, beneficios y alcance dentro del ciclo de QA.
  - Repaso de conceptos fundamentales de QA manual vs. automatizado (tipos de pruebas: unitarias, integración, end-to-end; pirámide de pruebas).
  - Configuración del entorno de desarrollo en Python: instalación de Python y pip, elección de un IDE (VSCode, PyCharm), creación de entorno virtual.
  - Control de versiones con Git y GitHub: flujo de trabajo básico (init, clone, commit, push, pull) y organización del repositorio para el bootcamp.

códigofacilito

Plan de estudio

- **Clase 2: Primeros pasos con Python aplicado a pruebas**
  - Repaso de sintaxis básica de Python (variables, tipos de datos, estructuras de control) orientado a su uso en scripts de testing.
  - Fundamentos de programación orientada a objetos en Python (clases, métodos) con ejemplos sencillos relacionados a pruebas.
  - Introducción a PyTest: instalación de la librería, estructura básica de un test (nomenclatura de archivos y funciones de prueba).
  - Creación y ejecución del primer test con PyTest (por ejemplo, probar una función sencilla como calculadora) y análisis del resultado en consola.

códigofacilito

Plan de estudio

- Práctica Semana 1: Cada alumno configura su entorno de desarrollo e inicializa un repositorio Git para el proyecto del bootcamp. Escribir un test básico usando PyTest para una función simple (por ejemplo, suma de dos números) y ejecutar el test correctamente.

- Reto Semana 1: Documentar brevemente un caso real donde la automatización de pruebas aporte valor frente a pruebas manuales. Adicionalmente, crear un script corto en Python que lea datos de un archivo (JSON o CSV) y muestre un resultado, para reforzar la familiarización con Python.

códigofacilito

Plan de estudio

Semana 2: Fundamentos de Pruebas Automatizadas con PyTest

- **Clase 1: Uso intermedio de PyTest – estructura, aserciones y fixtures**
  - Anatomía de un test en PyTest: reglas de descubrimiento de pruebas (nombres de archivos test_*.py), funciones de setup/ teardown implícitas.
  - Uso de aserciones en Python/PyTest para validar resultados esperados vs. obtenidos.
  - Introducción a fixtures en PyTest: definición de fixtures simples (por ejemplo, datos de prueba o configuración) y alcance (function, module).
  - Marcas (markers) en PyTest: categorizar pruebas (ej. @pytest.mark.smoke, @pytest.mark.regression) y ejecución selectiva con opciones de línea de comando.
- **Clase 2: Buenas prácticas en diseño de tests y parametrización**
  - Principio AAA (Arrange-Act-Assert) para estructurar claramente el código de prueba.
  - Parametrización de pruebas con PyTest (@pytest.mark.parametrize) para probar múltiples conjuntos de datos sin duplicar código.

códigofacilito

Plan de estudio

Semana 2: Fundamentos de Pruebas Automatizadas con PyTest

- Manejo de excepciones en tests: verificar que se lancen errores esperados con pytest.raises.
- Organización del código de pruebas vs. código de aplicación: cómo estructurar proyectos de tests (separación de utilidades, datos, fixtures compartidos).
- Interpretación de resultados y reportes básicos: uso de opciones como -v/-vv para ver detalles de tests y preparación para generar reportes más adelante.

- Práctica Semana 2: Escribir un conjunto de casos de prueba con PyTest para una biblioteca de funciones proporcionada (ejemplo: funciones utilitarias de cadenas de texto o cálculo matemático). Incluir al menos un fixture para preparar datos y usar parametrización en varios tests para cubrir múltiples escenarios con menos código repetido.
- Reto Semana 2: Diseñar y implementar un pequeño suite de pruebas para una aplicación sencilla (se puede proporcionar código base, ej. una calculadora de impuestos). Debe incluir: casos positivos y negativos, uso de fixtures (por ejemplo, setup de entorno o datos), y parametrización donde aplique. Ejecutar la suite localmente y subir los resultados (por ejemplo, captura de pantalla o log de consola) al repositorio GitHub.

códigofacilito

Plan de estudio

Semana 3: Pruebas de API – Introducción a Requests y PyTest

- **Clase 1: Fundamentos de pruebas de API REST**
  - ¿Qué es una API REST? Repaso de conceptos HTTP: métodos (GET, POST, PUT, DELETE), códigos de estado (200, 404, 500...), cabeceras y cuerpo (JSON).
  - Introducción a la librería Requests de Python: realización de peticiones GET/POST a una API pública (ejemplo: JSONPlaceholder u otra API de ejemplo) y compr
  - Análisis de respuestas JSON: cómo parsear JSON en Python (módulo json o métodos de Requests) y extraer valores para verificación.
  - Verificaciones básicas en APIs: validar código de estado esperado, campos obligatorios en la respuesta y valores retornados correctos.
- **Clase 2: Automatización de pruebas de API con PyTest**
  - Estructura de pruebas de API en PyTest: organización de tests API separados de tests unitarios, uso de fixtures para datos comunes (p. ej., URL base de la API, autenticación si es necesaria).

códigofacilito

Plan de estudio

- Escritura de casos de prueba API usando Requests dentro de funciones de PyTest (por ejemplo, probar que un GET a un recurso existente devuelve 200 y ciertos datos, y que un GET a un recurso inexistente devuelve 404).
- Validación de distintos aspectos de la respuesta: cuerpo JSON correcto, tipos de datos esperados, tiempo de respuesta dentro de un umbral razonable (introducción ligera a performance).
- Uso de parametrización para pruebas de API (ej.: probar múltiples endpoints o múltiples entradas en un mismo endpoint de forma eficiente).
- Buenas prácticas en pruebas de API: pruebas independientes entre sí (cada test prepara su propio contexto), evitar dependencias fuertes de datos (crear y borrar entidades de prueba en cada ejecución si es posible).
- Práctica Semana 3: Utilizar Requests y PyTest para escribir pruebas contra una API de ejemplo. Por ejemplo, consumir la API pública JSONPlaceholder: verificar que al hacer GET a /posts retorna una lista de 100 posts, que un GET a /posts/1 contiene campos específicos (userId, id, title, body), y que se puede crear un nuevo post con POST obteniendo un código 201.

códigofacilito

Plan de estudio

- Reto Semana 3: Ampliar las pruebas de API para cubrir escenarios de error y bordes: por ejemplo, verificar que al pedir un recurso inexistente la API responde con código correcto y mensaje de error, o que no se puede crear un recurso con datos faltantes (y la API devuelve código 400 o similar). Incluir autenticación en las pruebas si la API lo requiere (o simular llamada a un endpoint seguro con token ficticio) para demostrar manejo de headers/token. Documentar en el README del repositorio cómo ejecutar estas pruebas de API.

Semana 4: Pruebas de API – Casos Avanzados y Mini-framework

- **Clase 1: Pruebas avanzadas de API y manejo de datos dinámicos**
  - Autenticación en APIs: introducción a métodos de auth (API keys, tokens JWT, OAuth 2.0) y ejemplo de uso de Requests con headers o tokens (por ejemplo, incluir un token Bearer en los headers de la petición).
  - Flujo de prueba dependiente: crear datos de prueba en una petición (POST) y usarlos en pruebas posteriores (GET/PUT), asegurando limpiar o aislar dichos datos para no afectar ejecuciones futuras.

códigofacilito

Plan de estudio

- Validación de esquemas de respuesta: concepto de verificar que la estructura del JSON cumple con lo esperado (mencionando herramientas como jsonschema para validación, aunque sea opcional).
- Manejo de paginación o lotes en APIs (si aplica): cómo probar endpoints que devuelven datos paginados o listas grandes de forma eficiente.
- Consejos para evitar falsos negativos/positivos en tests API: reintentos simples ante fallos intermitentes, uso adecuado de tiempos de espera o esperas entre llamados si el sistema requiere tiempo para procesar.

- **Clase 2: Construcción de un mini framework de pruebas de API**
  - Refactorización: organizar el código de pruebas API en funciones reutilizables o módulos. Por ejemplo, crear una función helper para hacer peticiones GET/POST que incluya manejo de URL base y headers comunes.
  - Estructura del proyecto de tests API: dividir pruebas por módulo o funcionalidad de la API, mantener datos de configuración (URLs, credenciales) en un solo lugar (archivos de configuración o variables de entorno).

códigofacilito

Plan de estudio

- Refactorización: organizar el código de pruebas API en funciones reutilizables o módulos. Por ejemplo, crear una función helper para hacer peticiones GET/POST que incluya manejo de URL base y headers comunes.
- Estructura del proyecto de tests API: dividir pruebas por módulo o funcionalidad de la API, mantener datos de configuración (URLs, credenciales) en un solo lugar (archivos de configuración o variables de entorno).
- Práctica Semana 4: Reestructurar las pruebas de API desarrolladas para aplicar refactorización: centralizar la URL base de la API en un fixture o archivo de config, y crear funciones auxiliares si hay código repetitivo en las llamadas. Ejecutar el suite completo y generar un reporte HTML utilizando pytest-html u otro plugin, verificando que incluya todos los casos.
- Reto Semana 4: Documentar y diseñar un plan de pruebas para un servicio web simulado (por ejemplo, una API de gestión de tareas o un módulo de carrito de compras). No es necesario implementarlo completamente, pero sí delinear los casos de prueba clave, incluyendo cómo se podrían implementar de forma automatizada. Implementar al menos 1-2 casos de ese plan en código real como demostración (pueden ser llamadas ficticias o a una API existente similar). Esto ayudará a practicar el diseño antes del proyecto final.

códigofacilito

Plan de estudio

Semana 5: Automatización Web – Introducción a Selenium WebDriver

- **Clase 1: Fundamentos de automatización web con Selenium**
  - ¿Qué es Selenium WebDriver? Arquitectura y componentes: cliente WebDriver, drivers específicos de navegador (ChromeDriver, GeckoDriver, etc.) y navegadores soportados.
  - Configuración inicial: instalación del paquete selenium en Python y descarga/configuración del WebDriver para Chrome o Firefox. Configuración de variables de entorno para el driver si aplica.
  - Navegación básica con Selenium: abrir un navegador con webdriver.Chrome(), cargar una URL de prueba.
  - Estrategias de localización de elementos: identificadores comunes (ID, name, class, CSS selector, XPath) y métodos find_element/s para obtener elementos de la página.
  - Interacciones simples: hacer click en botones/ enlaces, ingresar texto en campos (send_keys), limpiar campos, obtener texto de elementos y validar valores mostrados.

códigofacilito

Plan de estudio

- **Clase 2: Selenium + PyTest – Estructurando pruebas web automatizadas**
  - Integración de Selenium en pruebas PyTest: uso de fixtures para iniciar el navegador antes de cada prueba y cerrarlo al final (garantizando limpieza independientemente de si la prueba pasa o falla).
  - Manejo de esperas en la web: diferencias entre esperas implícitas y esperas explícitas. Uso de WebDriverWait y condiciones esperadas (ExpectedConditions) para esperar que elementos aparezcan/cumplan cierta condición antes de interactuar.
  - Ejemplo de prueba completa: automatizar un caso sencillo (por ej., escenario de login en un sitio demo) combinando pasos: abrir página, ingresar credenciales, hacer click e inspeccionar un resultado (URL de redirección o mensaje de bienvenida) con una aserción.
  - Gestión de estados antes y después de pruebas: por ejemplo, asegurarse de estar deslogueado antes de probar login, o volver a estado inicial tras la prueba (logoff).
  - Solución de problemas comunes: manejar ventanas emergentes o alertas (método switch_to.alert), manejar múltiples pestañas/ventanas si ocurre, y captura de pantalla en caso de fallo para depuración.

códigofacilito

Plan de estudio

- **Práctica Semana 5: Desarrollar una prueba automatizada con Selenium que verifique una funcionalidad básica en una página web de prueba. Por ejemplo: navegar a una página de registro de usuario, completar el formulario con datos de ejemplo y verificar que aparece un mensaje de éxito o que se redirige a la pantalla esperada. La prueba debe estructurarse en PyTest, usando un fixture para inicializar/cerrar el navegador, e incluir esperas explícitas donde sea necesario.**
- **Reto Semana 5: Parametrizar la prueba anterior para ejecutarla con diferentes datos (ej.: varios conjuntos de datos de registro, incluyendo alguno inválido para comprobar validaciones de formulario). Asegurarse de manejar correctamente cada caso (limpiar entre iteraciones si el estado persiste, como borrar cookies o usar un navegador nuevo por iteración). Documentar cualquier desafío encontrado, como elementos difíciles de identificar o sincronización necesaria.**

Semana 6: Automatización Web – Page Object Model y Pruebas E2E

- **Clase 1: Page Object Model (POM) y mantenibilidad de pruebas UI**
  - Introducción al patrón Page Object Model: explicación de cómo representar páginas web como clases en Python, para desacoplar la lógica de localización de elementos de las pruebas mismas.
  - Implementación de un POM básico: crear una clase para una página (ej: LoginPage) con atributos para selectores (ej. self.username_input) y métodos para interacciones (ej. login(usuario, pass) que rellena formulario y envía).

códigofacilito

Plan de estudio

- Refactorización: convertir las pruebas existentes de la semana 5 para usar el POM en lugar de llamadas directas de Selenium en cada test. Ver cómo mejora la legibilidad y reutilización del código.
- Interacciones avanzadas con Selenium: uso de ActionChains para acciones complejas (arrastrar y soltar, hover sobre elementos, doble click) y manejo de elementos dinámicos (elementos que aparecen/disaparecen con retraso, usando nuevamente esperas explícitas cuando haga falta).
- Captura de evidencias: tomar screenshots automatizados en puntos clave o al final de pruebas, y especialmente al ocurrir una excepción, para facilitar el análisis de fallos en pruebas UI.

- **Clase 2: Pruebas end-to-end y multi-navegador**
  - Construcción de casos de prueba end-to-end que cubren flujos completos de la aplicación web. Ejemplo: "El usuario realiza una compra en el sitio" abarcando desde login, navegación de catálogo, agregar al carrito y checkout (simulado si no hay transacción real).
  - Manejo de múltiples páginas en un flujo: utilizar varios objetos POM (ej: HomePage, ProductPage, CartPage) en secuencia dentro de un mismo test para reflejar un caso de uso real.

códigofacilito

Plan de estudio

- Ejecución de pruebas en diferentes navegadores: configurar WebDriver para otro navegador (Firefox, Edge) y comentar sobre correr pruebas en modo headless (sin interfaz gráfica) para entornos de CI.
- Introducción a pruebas en paralelo para UI (concepto): dificultades de ejecutar varias instancias de navegador simultáneamente y recomendaciones (generalmente ejecutar en serie o usar grids/servicios en la nube para escalar).
- Mejores prácticas de mantenimiento: actualizar selectores de POM cuando cambia la UI, mantener datos de prueba separados (por ejemplo, en archivos YAML/JSON) para facilitar cambios, y limitar las pruebas E2E a los casos críticos para mantener la suite rápida.
- Práctica Semana 6: Implementar el Page Object Model para al menos una de las páginas del flujo de pruebas (por ejemplo, la página de Login o la de Registro) e integrar su uso en las pruebas existentes. Asegurarse de que todas las pruebas siguen pasando con la nueva estructura. Luego, crear una nueva prueba E2E que involucre navegar por varias páginas (utilizando POMs de cada una) – por ejemplo, buscar un producto y verificar que aparece en resultados, luego ir al detalle del producto.
- Reto Semana 6: Adaptar la suite de pruebas web para ejecutarse en dos navegadores diferentes (p.ej., Chrome y Firefox). Esto puede implicar configurar dos fixtures de WebDriver o utilizar parametrización para el navegador. Ejecutar la suite completa en ambos navegadores y anotar cualquier diferencia observada. Además, ejecutar las pruebas en modo headless para comprobar que siguen funcionando en ausencia de interfaz gráfica. Documentar los ajustes realizados para soportar multi-navegador y headless.

códigofacilito

Plan de estudio

Semana 7: Pruebas BDD con Cucumber (Behave)

- **Clase 1: Introducción a BDD y escritura de Features**
  - Conceptos de Behavior Driven Development (BDD): colaboración entre equipos (QA, dev, negocio) para definir comportamientos esperados antes de implementar.
  - Lenguaje Gherkin: sintaxis para escribir escenarios en lenguaje natural estructurado. Palabras clave: Feature, Scenario, Given/When/Then (en español: Característica, Escenario, Dado/Cuando/Entonces).
  - Ejemplo práctico: crear una especificación Gherkin para una funcionalidad simple (ej: autenticación) describiendo escenario de login exitoso y fallido en texto entendible por todos.
  - Herramienta Cucumber en Python: presentación de la librería Behave (framework BDD en Python equivalente a Cucumber) que interpreta archivos .feature. Estructura de un proyecto Behave: directorios features con archivos .feature y pasos (steps/ con step definitions en Python).
  - Sintaxis de step definitions en Behave: cómo mapear frases de Gherkin a funciones Python usando expresiones regulares o frases parseables (decoradores @given, @when, @then).

códigofacilito

Plan de estudio

- **Clase 2: Implementación de pruebas BDD con Selenium y Requests**
  - Implementar escenarios BDD con código: tomar el feature de ejemplo (ej: login) y escribir las step definitions usando Selenium para la parte UI (ej.: en el paso Cuando el usuario ingresa credenciales válidas y envía, automatizar ese flujo con Selenium). Si hubiera pasos de API, usarlos con Requests de manera similar.
  - Ejecutar las pruebas BDD con behave y analizar el resultado (pasos ejecutados, escenarios pasados o fallidos). Comparar la salida con la de PyTest y ver cómo Behave destaca cada paso.
  - Discusión: Ventajas de BDD (documentación viva, claridad para el negocio) vs. complejidad añadida. Buenas prácticas al escribir escenarios (evitar demasiados detalles técnicos en el texto, centrarse en el qué y no el cómo).
  - Integración de BDD con nuestro framework: cómo podemos coexistir o integrar escenarios BDD con nuestras pruebas PyTest. Mención de alternativas como pytest-bdd (plugin de PyTest para Gherkin) como otro enfoque.
  - Ejercicio adicional: escribir uno o dos escenarios más para otra funcionalidad pequeña en formato Gherkin y discutir en grupo cómo se implementarían los steps correspondientes.

códigofacilito

Plan de estudio

- Práctica Semana 7: Crear un archivo de característica (.feature) con al menos un Feature y dos Scenarios describiendo una funcionalidad (por ejemplo: búsquedas en la página web de prueba o creación de un recurso vía API). Implementar los step definitions necesarios en Python usando Selenium o Requests según corresponda a cada paso. Ejecutar behave para correr estos escenarios y asegurarse de que pasan correctamente.
- Reto Semana 7: Ampliar el conjunto de pruebas BDD: por ejemplo, describir todo un flujo de negocio en Gherkin (varios escenarios encadenados lógicamente, como el proceso de compra desde agregar al carrito hasta confirmación). Implementar los pasos clave de al menos un escenario complejo. Considerar el uso de hooks de Behave (@Before/After) para setup/teardown en contextos BDD (ej.: abrir/cerrar navegador antes de cada escenario). Documentar cómo estas pruebas BDD podrían integrarse en el proyecto final y qué valor añaden para stakeholders no técnicos.

Semana 8: Integración Continua (CI) y Entrega Continua (CD) Básicas

- **Clase 1: Conceptos de CI/CD y pipelines de testing**
  - Introducción a Integración Continua (CI): concepto de ejecutar procesos automáticos con cada cambio de código. Beneficios para QA (detección temprana de fallos).
  - Introducción a Entrega/Despliegue Continuo (CD): concepto de liberar software de forma frecuente y controlada, y rol de las pruebas automatizadas en garantizar calidad en cada despliegue.
  - Herramientas populares de CI/CD: visión general de Jenkins, Travis CI, GitLab CI, GitHub Actions, etc., y cómo se configuran mediante pipelines (.yml, Jenkinsfile).

códigofacilito

Plan de estudio

- Creación de un pipeline básico de CI: pasos típicos (checkout de código, instalación de dependencias, ejecución de pruebas, recopilación de resultados). Ejemplo de configuración con GitHub Actions para ejecutar pytest automáticamente en cada push o pull request.
- Notificaciones de resultados: cómo los pipelines informan de éxito/fallo (por ejemplo, estatus en GitHub, emails, Slack) y la importancia de una retroalimentación rápida al equipo.

- **Clase 2: Implementación práctica de CI para la suite de pruebas**
  - Configuración de un pipeline CI real en el proyecto del bootcamp: escribir un archivo YAML de GitHub Actions que instale Python, instale requerimientos (Selenium, Requests, etc.), y ejecute la suite completa de PyTest.
  - Manejo de particularidades en CI para pruebas UI: por ejemplo, usar modo headless para Selenium en servidores Linux, instalar drivers de navegador o usar servicios como Chrome Headless.

códigofacilito

Plan de estudio

- Introducción conceptual a CD: por ejemplo, demostrar cómo se podría extender el pipeline para desplegar una app en un entorno de prueba una vez pasen los tests (aunque no se implementará por completo, dar la idea de continuidad hacia deploy).

- Práctica Semana 8: Implementar la integración continua en el repositorio del proyecto: agregar un pipeline que al hacer push ejecute los tests de API y de UI. Probar el pipeline haciendo un commit y verificando en la plataforma CI que las pruebas corren y dan feedback (verde/rojo). Ajustar cualquier fallo de entorno (por ej., falta de algún paquete, path del WebDriver) hasta lograr una ejecución exitosa.

- Reto Semana 8: Mejorar el pipeline de CI incorporando un paso de generación de reporte de resultados (por ejemplo, subir el archivo HTML de reporte de PyTest como artefacto, o integrar Allure Report para tener reportes más detallados). Documentar en el repositorio las configuraciones de CI realizadas.

códigofacilito

Plan de estudio

Sesión Intermedia: Proyecto Final (Desarrollo del Framework de Pruebas)

- **Clase 1: Planificación y diseño del proyecto final**
  - Presentación del Proyecto Final: descripción del sistema o aplicación a probar (por ejemplo, una aplicación web demo con su API de backend – podría ser un e-commerce simplificado, una app de gestión de tareas, etc.). Se detallan los alcances y requisitos de pruebas esperados.
  - Definición del alcance de las pruebas: identificar las funcionalidades críticas a automatizar tanto en la capa API como en la UI (ej.: creación de usuarios, login/logout, operaciones CRUD de un recurso, flujo de compra completo, etc.).
  - Arquitectura del framework de pruebas: decidir estructura de carpetas (tests API, tests UI, page objects, fixtures comunes, data, etc.), convenciones de nombres y uso de herramientas (por ejemplo, si se usará Behave para ciertos casos BDD dentro del proyecto o solo PyTest).
  - Estrategia de trabajo en equipo (si el proyecto es grupal) o individual: distribución de tareas, integración continua colaborativa (ramas de Git, pull requests para merge).
  - Revisión de checklist inicial: confirmar que todos tienen su entorno listo, acceso a la aplicación a probar, y recordar los criterios de evaluación del proyecto (calidad de pruebas, cobertura, uso de herramientas, etc.).


códigofacilito
Apertura de inscripciones:
10 de abril 2025
Inicio de clases:
19 de Junio 2025

códigofacilito

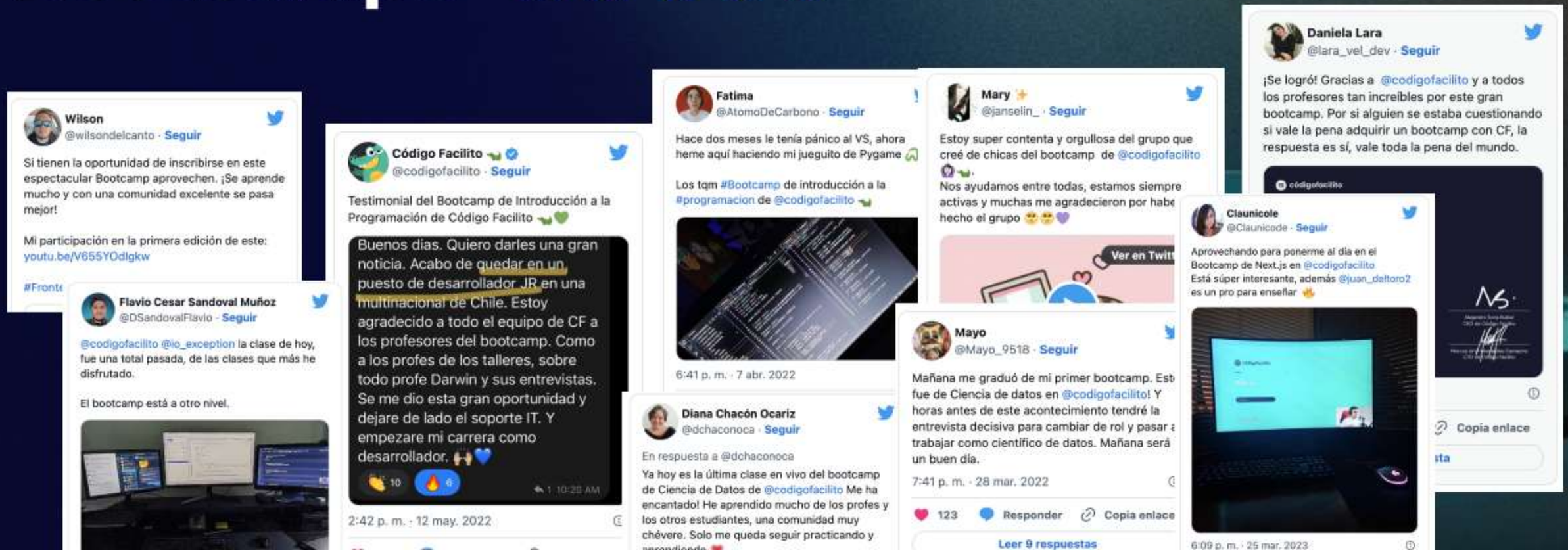
*Clases Martes y jueves:

5 pm - 🇲🇽 🇬🇹 🇸🇻 🇨🇷 🇸🇻
6 pm - 🇨🇴 🇵🇪 🇨🇺 🇵🇷
7 pm - 🇻🇪 🇧🇴 🇨🇱 🇦🇺
8 pm - 🇦🇷
12 am - 🇪🇸

**No es requisito asistir en vivo: puedes seguir el Bootcamp a tu ritmo ya que todas las clases quedan grabadas.*

códigofacilito

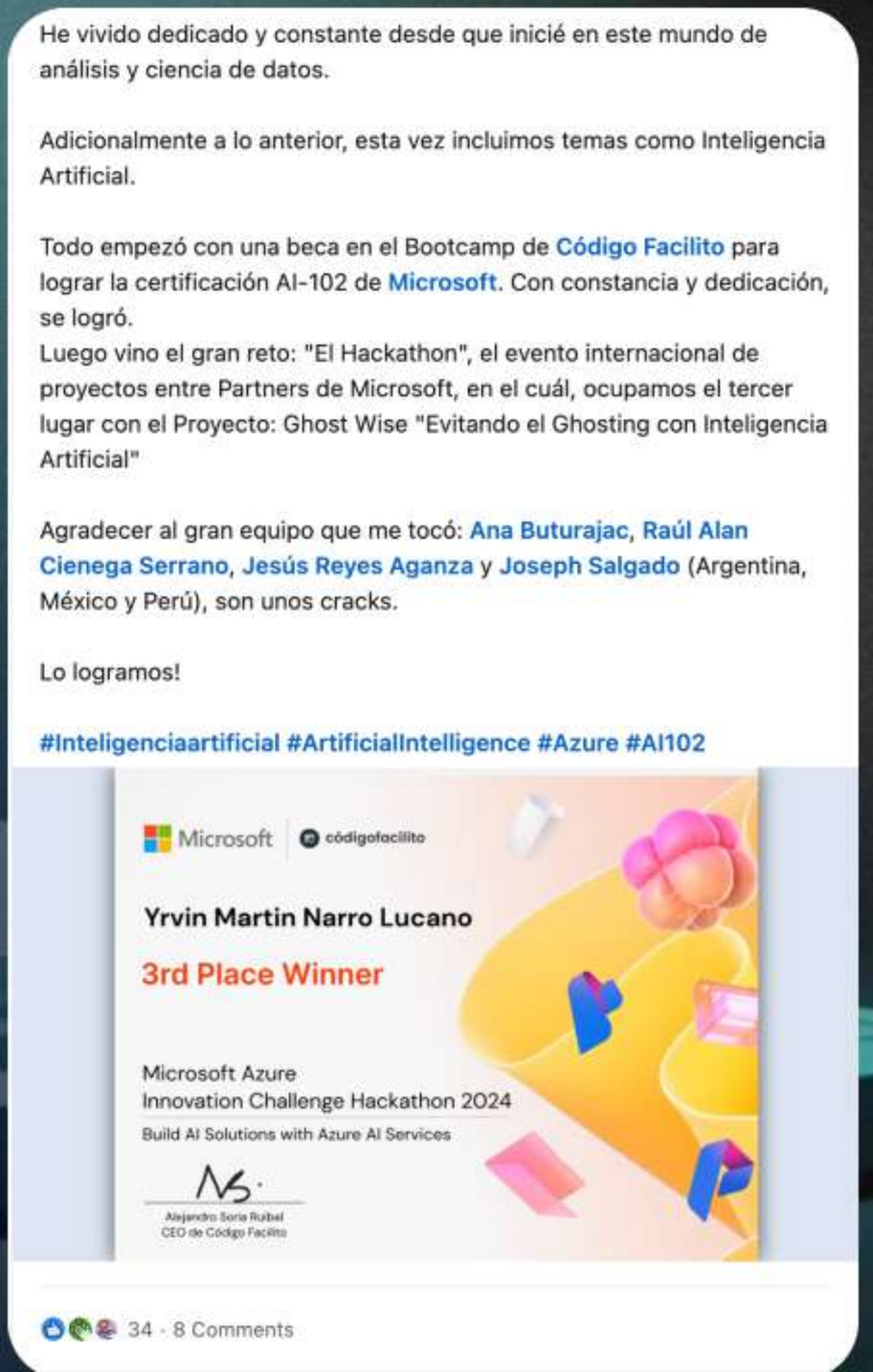
Lo que dicen de nuestros Bootcamps ★★★★★



códigofacilito

Testimonios del éxito de nuestros estudiantes en Linkedin

★★★★★



códigofacilito

Prueba 2 clases:

Si las primeras 2 clases no superan tus expectativas, ofrecemos un **reembolso completo** de tu dinero sin preguntas.

Esta es nuestra garantía de calidad.

★★★★★

códigofacilito

¡Estás a un paso de sumarte al Bootcamp!

codigofacilito.com/**bootcamps/testing-automatizado**

¿Alguna duda?

ayuda@codigofacilito.com