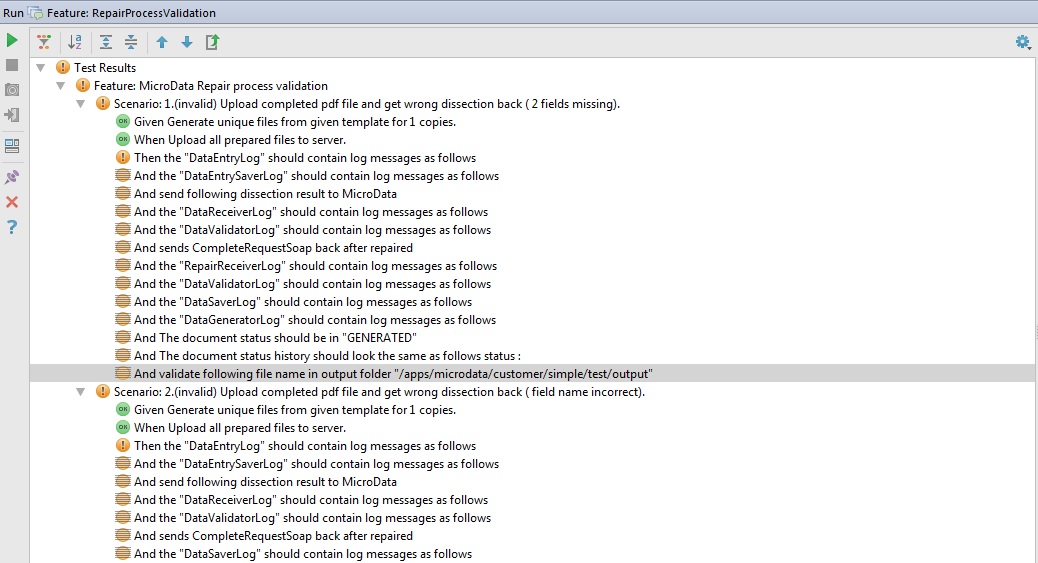
**บทที่ 4**

**ผลการดำเนินงาน**

หลังจากศึกษาและพัฒนาซอฟต์แวร์ทดสอบอัตโนมัติ พร้อมกับสร้างกรณีทดสอบที่ครอบคลุมการทำงานโดยรวมของระบบ MicroData เป็นอันเสร็จสิ้น ทางทีมพัฒนาระบบ MicroData ก็ได้นำเอาซอฟต์แวร์ทดสอบอัตโนมัติไปใช้งานจริงในการทดสอบระบบ ซึ่งผลลัพธ์ที่ได้จากการนำเอาซอฟต์แวร์ทดสอบอัตโนมัติไปใช้ทำการทดสอบระบบนั้น พบว่าสามารถช่วยให้ตรวจพบข้อผิดพลาดของระบบที่เกิดขึ้นในระหว่างการพัฒนา ซึ่งช่วยให้ทีมพัฒนาสามารถแก้ไขปัญหา และข้อผิดพลาดที่เกิดขึ้น ได้รวดเร็วและทันท่วงที ดังนั้นการใช้ซอฟต์แวร์ทดสอบทุกครั้งที่มีการเปลี่ยนแปลงแก้ไขซอร์สโค้ดของระบบ จึงช่วยเพิ่มความมั่นใจให้กับทีมพัฒนาว่า หากซอร์สโค้ดที่เขียนขึ้นผ่านการทดสอบด้วยซอฟต์แวร์อัตโนมัติแล้ว จะมั่นใจได้ว่าไม่มีปัญหาเกิดขึ้นกับระบบเดิมที่เคยทำงานได้ ดังนั้นจึงได้มีการรวมซอฟต์แวร์เข้าสู่ระบบ Continuous Integration ซึ่งเป็นระบบที่ช่วยให้ทุกครั้งที่มีการเปลี่ยนแปลงซอร์สโค้ดหลักของระบบแล้วจะมาทำการสั่งรันเคสทอดสอบของซอฟต์แวร์โดยอัตโนมัติ ทำให้ทีมพัฒนาไม่ต้องมาสั่งรันเคสทดสอบเอง นอกจากนั้นการทดสอบยังใช้เวลาค่อนข้างสั้น ดังนั้นทัมพัฒนาจึงสามารถทำการรันเคสทดสอบได้เรื่อยๆ ในระหว่างการพัฒนา นอกจากนั้นแล้วทีมพัฒนายังสามารถนำเอาบาง step ที่พัฒนาขึ้นภายในซอฟต์แวร์ไปประยุกต์ใช้ในการอัพโหลดไฟล์จำนวนมากขึ้นสู่ระบบ เพื่อใช้ในการทดสอบวัดประสิทธิภาพได้อีกด้วย

**4.1 ผลการพัฒนาซอฟต์แวร์**

สำหรับผลลัพธ์ของการรันกรณีทดสอบหรือ scenario ภายใต้ feature file นั้น จะทำการรันด้วย IntelliJ IDE ซึ่งเป็นซอฟต์แวร์ที่ใช้ช่วยในการเขียนโปรแกรม และสามารถทำการสั่งรัน feature file ได้โดยตรง โดยที่ไม่ต้องทำการเขียน test runner ขึ้นมาด้วย unit test เพื่อใช้ในการรัน feature file เหมือนกับการใช้ IDE ตัวอื่น โดย IntelliJ จะแสดงผลลัพธ์ของการทดสอบออกมาอยู่ในรูปของรายงานที่แสดงด้วยรูปภาพ ดังรูปตัวอย่างที่ 4.1



**รูปที่ 4.1** ตัวอย่างรายงานผลการทดสอบ

โดยจากรูปที่ 4.1 จะเป็นตัวอย่างของ รายงานผลการทดสอบของการรันกรณีทดสอบทั้งหมดภายใต้ feature RepairProcessValidation ที่ถูกสร้างขึ้นมาโดย IntelliJ โดยรายงานที่ถูกสร้างขึ้นมานั้นจะแสดงข้อมูลต่างๆดังต่อไปนี้

1. ชื่อของ Feature ที่ทำการทดสอบ
2. ชื่อของ Scenario ทั้งหมดภายใต้ Feature ที่ทำการรัน
3. ผลการทดสอบในแต่ละ step โดยจะประกอบไปด้วย 3 ระดับคือ

* ผลการทดสอบว่า step ทำงานได้ถูกต้อง ไม่มี Assertion fail หรือการทำงานผิดพลาดใดๆ จะแสดงด้วยรูปภาพวงกลมสีเขียวพร้อมคำว่า OK อยู่ด้านหน้าของชื่อ step ดังรูปที่ 4.2



**รูปที่ 4.2** ตัวอย่างรายงานผลการทดสอบถูกต้อง

* ผลการทดสอบว่า step ทำงานผิดพลาด ซึ่งอาจเกิดจาก Assert fail, เมทอดเกิดการ Throw Exception หรือการทำงานผิดพลาดใดๆ ผลการทดสอบในกรณีนี้จะแสดงด้วยรูปภาพวงกลมสีเหลืองพร้อมเครื่องหมายอัศเจรีย์หน้าชื่อ step ดังรูปที่ 4.3



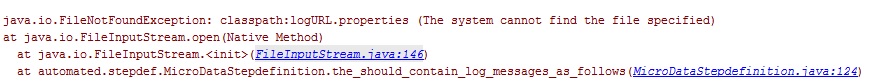
**รูปที่ 4.3** ตัวอย่างรายงานผลการทดสอบผิดพลาด

* ผลการทดสอบว่า step ไม่ถูกดำเนินการ เนื่องจากว่า step ก่อนหน้ามีการทำงานผิดพลาดหรือ step ไม่ได้มีส่วนของการ implementation ไว้ โดยจะแสดงผลด้วยรูปภาพวงกลมสีเหลืองพร้อมเส้นขีดแนวขวาง นำหน้าชื่อ step ดังรูปที่ 4.4



**รูปที่ 4.4** ตัวอย่างรายงานผลการทดสอบที่ถูกข้าม

โดยข้อมูลที่แสดงจากรูปที่ 4.1 บ่งบอกว่าการรัน feature ผิดพลาด โดยในแต่ละ scenario นั้น การทำงานของ step แรกและ step ที่สองนั้นถูกต้องส่วน step ที่สาม ซึ่งทำการตรวจสอบ processing log นั้นหาไม่เจอ จึงเกิดการตรวจสอบผิดพลาดที่ step ที่สามดังนั้น step อื่นๆที่มีขั้นตอนการทำงานต่อจาก step ที่สามทั้งหมดจึงถูกข้ามไป เพื่อไปทำการรัน scenario ถัดไปทันที ซึ่งในการใช้งานจริงนั้น เมื่อทีมพัฒนาทำการรัน feature เพื่อตรวจสอบความถูกต้องของระบบ เมื่อการรันสิ้นสุดและมาทำการดูรายงาน ก็จะสามารถทราบได้ว่า หากมีข้อผิดพลาดเกิดขึ้น ข้อผิดพลาดนั้นเกิดขึ้นที่ใด เกิดขึ้นเพราะอะไร ซึ่งจะช่วยให้ทำการแก้ปัญหาที่เกิดขึ้นกับระบบได้อย่างสะดวกและ รวดเร็ว โดยตัวช่วยที่ใช้ในการสังเกตุว่าข้อผิดพลาดเกิดจากอะไรนั้น ก็จะดูที่ console log ของ IntelliJ ซึ่งจะบอกว่า step ที่ผิดพลาดนั้น ผิดพลาดเพราะอะไร โดยหากผิดพลาดที่ ข้อความใน processing log ก็จะทำให้ทราบได้ว่าระบบผิดพลาดที่ข้อความไหน และช่วยให้สามารถไปแก้ไขได้ถูกจุด โดยตัวอย่างของข้อความแสดงข้อผิดพลาดใน step บน console log จะแสดงด้วยตัวอย่างในรูปที่ 4.5

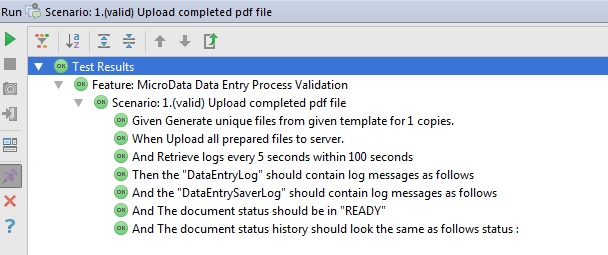


**รูปที่ 4.5** ตัวอย่างข้อความผิดพลาดใน console log

โดยจากรูปที่ 4.5 จะเป็นข้อความใน console log ที่บอกว่า step ที่ทำงานผิดพลาด ผิดพลาดเพราะหาไฟล์ที่เก็บ log ไม่เจอ ซึ่งอาจเกิดจากการลบไฟล์ผิด หรืออื่นๆ ซึ่งผู้ที่กำลังทำการทดสอบเมื่อเห็นข้อความก็จะสามารถไปแก้ไข สิ่งที่ผิดพลาดเพื่อให้การทดสอบทำได้อย่างถูกต้องจนได้ผลตามที่คาด โดยทางผู้เขียนจะแสดงตัวอย่างของกรณีที่การทดสอบทั้งหมดถูกต้อง และตัวอย่างของกรณีที่ขณะทดสอบมีข้อผิดพลาดเกิดขึ้น ดังต่อไปนี้

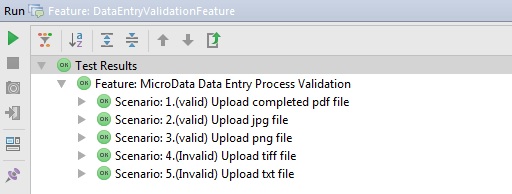
**4.1.1 ตัวอย่างผลการทดสอบกรณีที่การทดสอบถูกต้อง**

สำหรับผลการทดสอบในกรณีที่การทดสอบถูกต้อง หากการทดสอบนั้นเป็นการทดสอบ scenario จะหมายความว่า ทุกๆ step การทำงานภายใต้ scenario นั้นจะต้องทำงานได้อย่างถูกต้องครบทุก step ไม่มี step ที่ทำงานผิดพลาด โดยรายงานจะออกมาดังรูปที่ 4.6



**รูปที่ 4.6** รายงานผลการทดสอบกรณี scenario ถูกต้อง

โดยจะเห็นได้ว่าทุก step ใน scenario จะทำงานได้อย่างถูกต้อง และด้านบนสุดมีข้อความที่บ่งบอกว่าเป็น scenario อะไรที่ถูกทดสอบ, scenario นี้อยู่ใต้ feature ใด และผลลัพธ์เป็นอย่างไร ส่วนกรณีที่การทดสอบเป็นการทดสอบทั้ง feature หากทุก scenario ภายใต้ feature ผ่านการทดสอบอย่างถูกต้องทั้งหมด รายงานแสดงผลการทดสอบจะเป็นดังรูปที่ 4.7

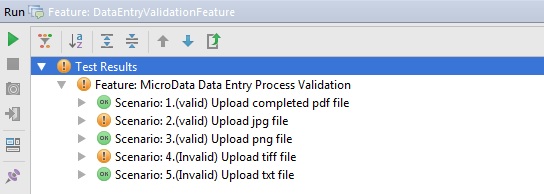


**รูปที่ 4.7** รายงานผลการทดสอบกรณีทั้ง feature ถูกต้อง

จากรูปจะเห็นได้ว่า รายงานจะแสดงโดยรวมถึงผลการทดสอบของแต่ละ scenario ว่าผลการทดสอบเป็นอย่างไร ซึ่งในกรณีนี้คือ ทุก scenario ภายใต้ feature นี้ทำงานได้ถูกต้องทั้งหมด ซึ่งเป็นการยืนยันได้ว่าระบบทำงานได้ถูกต้อง รวมถึงยังสามารถกดเข้าไปดูได้ว่า ในแต่ละ scenario ที่ทำนั้นประกอบไปด้วยการทำงานของ step ใดบ้าง นอกจากนี้แล้ว รายงานที่ถูกสร้างขึ้นยังช่วยให้รู้ได้ว่า scenario ใดภายใต้ feature ที่เราทำการทดสอบนั้นผิดพลาด และยังสามารถดูได้ว่า step ใดที่ทำให้การทดสอบผิดพลาด ซึ่งจะแสดงในหัวข้อถัดไป

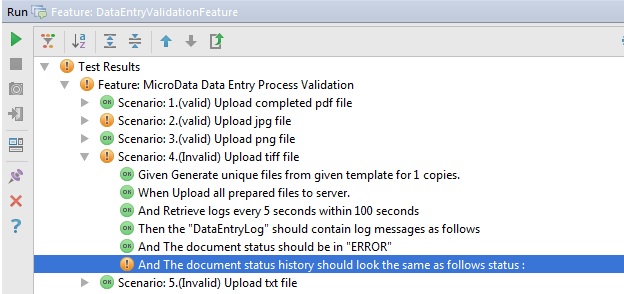
**4.1.2 ตัวอย่างผลการทดสอบกรณีที่การทดสอบผิดพลาด**

จากที่กล่าวมาในหัวข้อก่อนหน้า ในกรณีที่การทดสอบมีข้อผิดพลาดเกิดขึ้น รายงานที่ถูกสร้าง และแสดงออกมาโดย IntelliJ นั้นจะบ่งบอกว่ามีการทดสอบที่ผิดพลาดเกิดขึ้นกี่ scenario และเกิดขึ้นจาก scenario ใดบ้างใน feature รวมถึงยังสามารถกดเข้าไปดูเพิ่มเติมถึงรายละเอียดได้ว่า scenario ทีผิดพลาดนั้น เกิดจาก step ใด โดยตัวอย่างจะแสดงดังรูปที่ 4.8



**รูปที่ 4.8** ตัวอย่างรายงานแสดงผลในกรณีที่มี scenario ผิดพลาด

จากรูปจะเห็นได้ว่าเป็นการรันการทดสอบของ feature Data Entry process validation ซึ่งผลโดยรวมจะเห็นได้ว่า scenario ใดบ้างใน feature ที่ผ่านและ scenario ใดบ้างที่มีข้อผิดพลาดเกิดขึ้น ซึ่ง step ที่ทำให้ scenario ทดสอบผิดพลาดจะสามารถกดเข้าไปดูเพิ่มเติมถึงรายละเอียด step ที่ผิดพลาด ดังรูปที่ 4.9



**รูปที่ 4.9** ตัวอย่างรายงานแสดงผลข้อมูลเพิ่มเติมของ scenario ที่ผิดพลาด

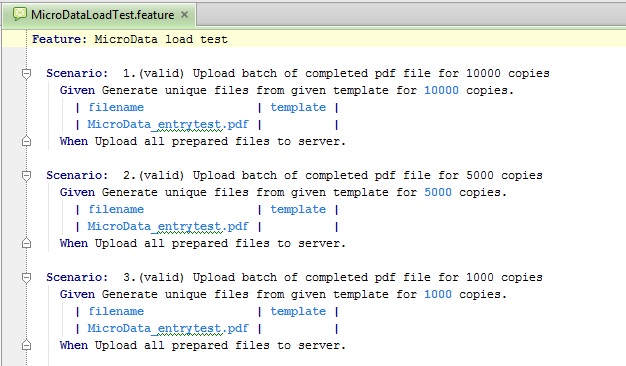
จากรูปที่ 4.9 จะเห็นได้ว่า step ที่ทำงานผิดพลาดก็คือ step ที่ทำหน้าที่ตรวจเช็คประวัติสถานะของไฟล์ ใน scenario ที่ 4 ซึ่งทำให้ผู้ที่ทำการรัน feature นี้เพื่อทดสอบระบบสามารถรู้ได้ว่าระบบผิดพลาดที่ step นี้และสามารถแก้ไขปัญหา โดยไปตรวจสอบที่ซอร์สโค้ดที่ทำงานเกี่ยวกับการดึงค่าสถานะ หรือไปตรวจสอบเพิ่มเติมว่าทำไมประวัติสถานะของไฟล์ถึงไม่ถูกต้อง เพื่อแก้ไขต่อไป

**4.2 การประยุกต์ใช้ซอฟต์แวร์ทดสอบอัตโนมัติ**

**4.2.1 การประยุกต์ใช้กับ Performance testing**

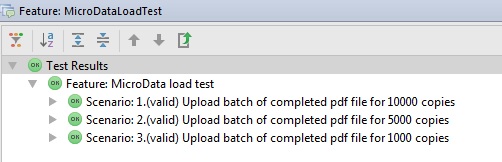
เนื่องจากว่า ในขณะที่สร้างซอฟต์แวร์ทดสอบอัตโนมัติ การพัฒนาระบบ MicroData กำลังอยู่ในช่วงการพัฒนาและ ปรับปรุงประสิทธิภาพของระบบ ดังนั้นทางทีมพัฒนาจึงต้องมีการทำ Performance testing หรือการทดสอบประสิทธิภาพของระบบ เพื่อให้รู้ว่า โปรเซสใดที่ต้องการการปรับปรุงประสิทธิภาพ เพราะฉะนั้น จึงต้องมีการทำการอัพโหลดไฟล์จำนวนมากเข้าสู่ระบบเพื่อจำลองถึงสถานการณ์การใช้งานจริง ที่ระบบถูกใช้งานอย่างหนักและต่อเนื่อง หรือการทำ load test ซึ่งหากเป็นการใช้งานจริงการนำเข้าไฟล์จำนวนมาก จะเกิดขึ้นจากทางฝั่งผู้ใช้ แต่เนื่องจากในการทดสอบ เราจึงจำเป็นต้องนำเข้าไฟล์เอง ซึ่งการทำ load test หากต้องให้นักทดสอบทำการอัพโหลดไฟล์เองทีละครั้ง จะเป็นการสิ้นเปลืองทรัพยากร และเวลาเป็นอย่างมากเนื่องจาก การทำ load test อาจจะต้องการนำเข้าไฟล์เป็นหลายรอบ แต่ละรอบนำเข้าด้วยปริมาณที่ต่างกันเป็นจำนวนมาก เช่น การแบ่งรอบนำเข้าที่ประกอบไปด้วย รอบละ 10,000 ไฟล์ 5,000 ไฟล์ และ 1,000 ไฟล์ ซึ่งเป็นไปไม่ได้ในการอัพโหลดด้วยมือ ดังนั้นทีมพัฒนาสามารถนำเอา step ส่วนต้นของซอฟต์แวร์ทดสอบอัตโนมัติ ซึ่งใช้ในการสร้างไฟล์จาก template และอัพโหลดไฟล์สู่ระบบ มาช่วยในการทำ load test ในส่วนนี้ ทำให้ประหยัดทรัพยากรเวลา และทำให้การทดสอบเป็นไปอย่างราบรื่น

โดยการทำ load test ด้วย ซอฟต์แวร์ทดสอบอัตโนมัตินั้น จะเริ่มจากการทำการสร้าง feature ขึ้นมาใหม่สำหรับการทำ load test ซึ่งภายใน feature จะประกอบไปด้วย scenario ที่ทำหน้าที่ในการสร้างไฟล์ขนาดต่างๆกันในแต่ละรอบ และอัพโหลดไฟล์ทั้งหมดที่สร้างขึ้นสู่ระบบ โดย feature ในการทำ load test นี้จะมีชื่อว่า MicroDataLoadTest feature ดังรูปที่ 4.10



**รูปที่ 4.10** MicroDataLoadTest feature

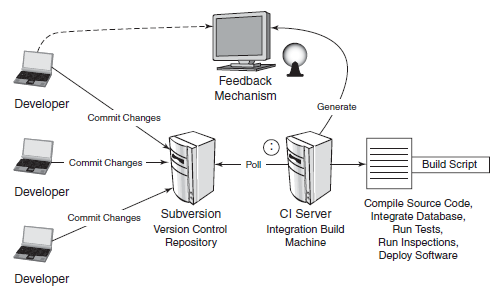
โดยในการใช้งานนั้น ผู้ทดสอบก็จะทำการรันทั้ง feature MicroDataLoadTest นี้ ซึ่งซอฟต์แวร์ก็จะทำงานสร้างไฟล์และอัพโหลดไฟล์ไปทีละ scenario ซึ่งอาจจะต้องใช้เวลาซักระยะหนึ่งเนื่องจาก ต้องทำการอัพโหลดไฟล์จำนวนมาก เมื่อเสร็จสิ้นการทำงานของ feature จะแสดงดังรูปที่ 4.11 และเมื่อไปตรวจเช็คข้อมูลก็จะพบว่ามีไฟล์จำนวนมากถูกอัพโหลดสู่ระบบเรียบร้อย



**รูปที่ 4.11** รายงานผลการรัน feature Load test

**4.3 การรวมซอฟต์แวร์กับระบบ Continuous integration**

เนื่องจากซอฟต์แวร์ทดสอบอัตโนมัติที่สร้างขึ้น สามารถใช้ช่วยทีมพัฒนาในการตรวจสอบระบบได้อย่างมีประสิทธิภาพ ดังนั้นการที่จะต้องมาทำการสั่งรัน feature ทั้งหมดด้วยมือ หลังจากทำการพัฒนาซอร์สโค้ดแต่ละครั้ง จึงไม่ใช่ทางเลือกที่ดีที่สุด ทางผู้เขียนจึงได้นำเอาความสามารถของระบบที่เรียกว่า continuous integration เข้ามาช่วย โดยการทำงานของระบบ continuous integration นั้นจะอธิบายด้วยรูปที่ 4.12



**รูปที่ 4.12** แผนภาพการทำงานของระบบ Continuous integration

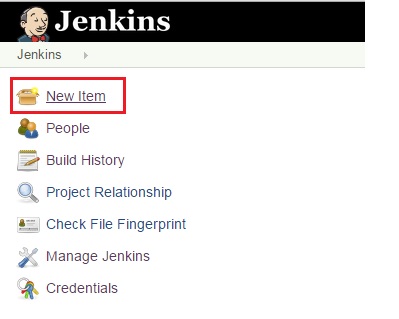
จากรูป การทำงานจะเริ่มจากเมื่อทีมพัฒนาทำการ commit changes หรือแก้ไขซอร์สโค้ดและนำกลับมารวมที่ระบบจัดการซอร์สโค้ดส่วนกลางเช่น Subversion, version control system จากนั้นจะมี CI server โดยในที่นี้เราจะใช้ CI server ที่มีชื่อว่า Jenkins โดย Jenkins นั้นจะมีหน้าที่ในการดูว่า เมื่อมีการเปลี่ยนแปลงซอร์สโค้ดหลักที่ version control system ก็จะไปทำการสั่งการทำงาน build script โดยอัตโนมัติ ซึ่ง build script ในที่นี้เช่น pom (Project object model) โดยในโปรเจคของเราจะมีไฟล์ pom อยู่ภายในโปรเจค ซึ่งในไฟล์ pom เราจะสามารถกำหนดได้ว่าให้ไปรัน unit test ใดบ้างในโปรเจคของเราเมื่อมีการ build เกิดขึ้น โดย Jenkins จะทำการสั่งรัน build script สำหรับงาน หรือ build job ที่เราได้สร้างเอาไว้ซึ่งใน build job ที่สร้างเราจะต้องกำหนดว่าเมื่อมีการสั่งงาน build job นี้จะให้ไปสั่งรัน pom ที่ใด ในที่นี้เราก็จะให้ Jenkins มารัน pom ของโปรเจคซอฟต์แวร์ทดสอบอัตโนมัติของเรา แต่ว่าใน build script จะไม่สามารถสั่งรัน feature file โดยตรงได้ เนื่องจาก pom จะสั่งรันได้เฉพาะ unit test ในโปรเจค ดังนั้นในกรณีของเราจะมีการเขียน unit test ขึ้นมาเรียกว่า feature test runner ซึ่งเอากำหนดว่าจะรัน feature file ใดบ้างในระบบหลังจากนั้นผลของการรัน build script จะส่งเป็น feedback กลับไปให้กับ developer ซึ่งในที่นี้อาจจะเป็น email แสดงผลหรือรายงาน ซึ่ง Jenkins จะสามารถแสดงรายงานผลการทดสอบในทุกๆครั้ง ซึ่งเราสามารถดูรายงานเหล่านี้ผ่านทางเว็บแอพพลิเคชั่น ของ Jenkins เองได้โดยตรง

ดังนั้นการนำเอาระบบ continuous integration มาประยุกต์ใช้สำหรับซอฟต์แวร์ของเรานั้น วิธีการก็คือ เราจะทำการสร้างงาน build job ภายใน Jenkins ให้กับซอฟต์แวร์ทดสอบอัตโนมัติของเราเอาไว้ หลังจากนั้นทุกๆครั้งที่ซอร์สโค้ดหลักของระบบ MicroData ถูก deploy เราจะเขียนคลาสในการรับฟัง (Listener Class) ในการเฝ้าสังเกตว่า เมื่อเกิดการ deploy ทุกครั้ง จะให้ทำการสั่งรัน งานที่เราเตรียมเอาไว้ให้ทำงาน เพื่อให้ทุกครั้งที่มีการเปลี่ยนแปลงในระบบ ซอฟต์แวร์ทดสอบอัตโนมัติของเราจะถูกสั่งรันทุก feature ที่ต้องการโดยอัตโนมัติ

**4.3.1 การสร้างงาน (build job) ด้วย Jenkins**

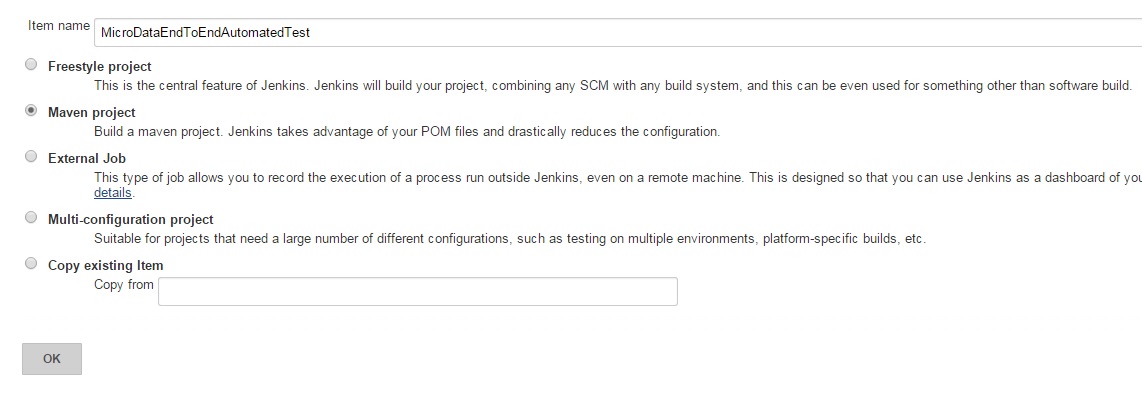
โดยขั้นตอนแรกในการทำงานกับเครื่องมือในการทำ continuous integration ที่เรียกว่า Jenkins นั้น ก่อนอื่นเราจะต้องทำการสร้างงาน หรือ build job ขึ้นมา ซึ่ง build job นั้นจะเป็นตัวเชื่อมกับโปรเจคที่ซอฟต์แวร์ของเราอยู่ เมื่อมีการสั่งรัน build job เกิดขึ้น Jenkins ก็จะไปทำการสั่งทำ unit test ทั้งหมดภายในโปรเจคของซอฟต์แวร์เรา ซึ่งรวมถึง feature test runner ด้วย

ขั้นตอนการสร้าง build job นั้นขั้นแรกให้เข้าไปที่ Jenkins สำหรับวงการพัฒนาของเราและเข้าสู่เมนู New Item ดังรูปที่ 4.13



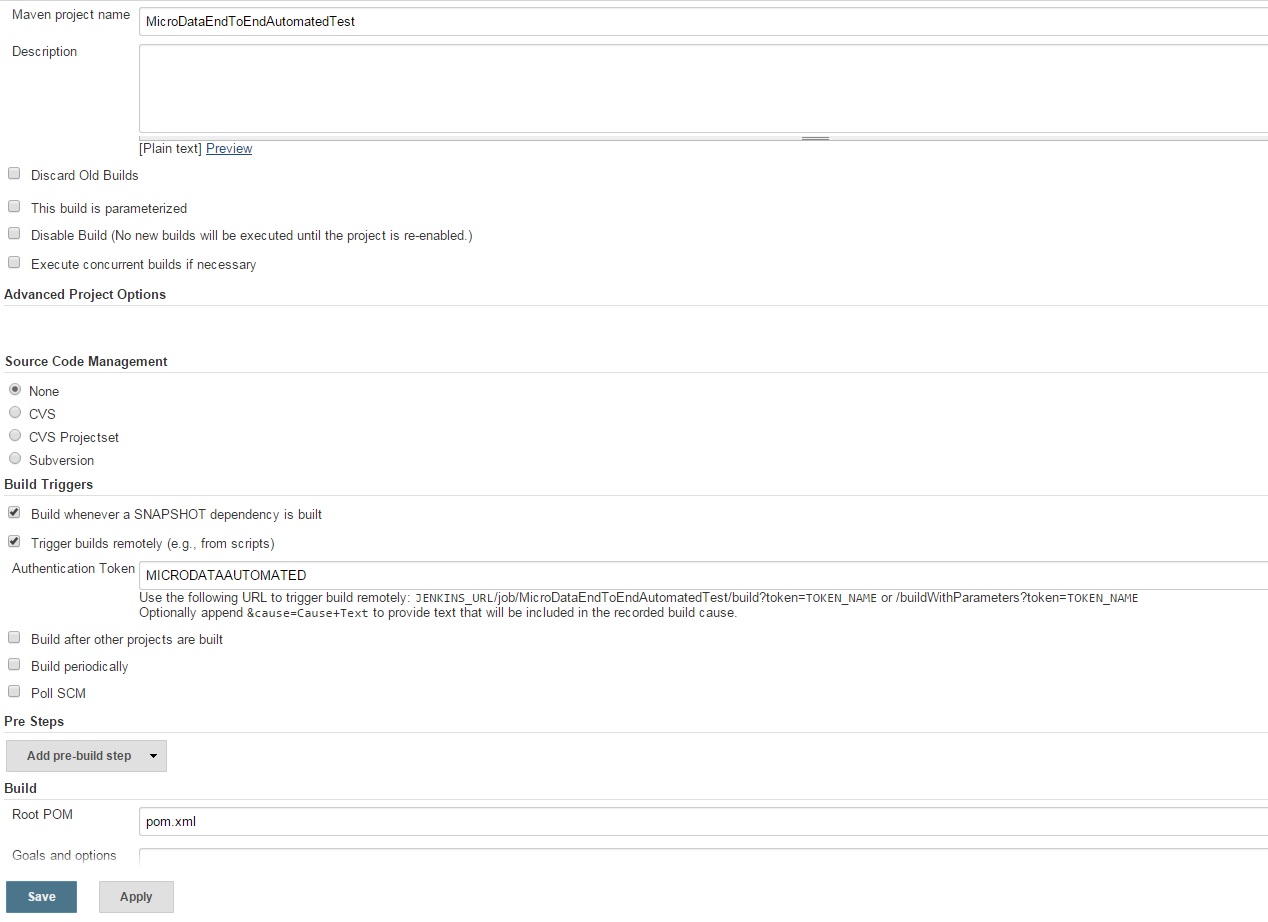
**รูปที่ 4.13** รูปเมนูหน้าแรกของ Jenkins

เมื่อเข้ามาแล้วจะแสดงเมนูดังรูปที่ 4.14



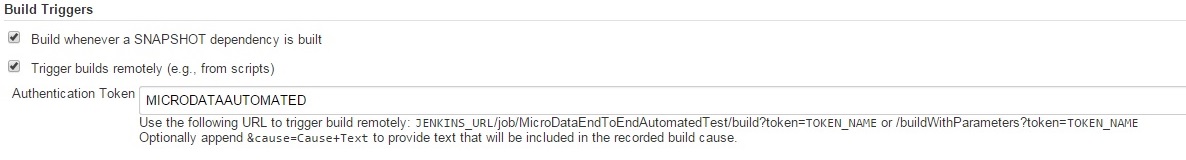
**รูปที่ 4.14** รูปเมนูประเภท build job ของ Jenkins

กรอกชื่อของ build job และเลือก Maven project เพื่อให้ job ทำการรันกับไฟล์ pom ในโปรเจค แล้วกด OK เพื่อไปหน้าการตั้งค่า build job ในรูปที่ 4.15



**รูปที่ 4.15** รูปเมนูการตั้งค่า build job ของ Jenkins

การตั้งค่าให้ทำการเลือกตัวเลือก trigger builds remotely พร้อมตั้งค่า token เพื่อให้ build job นี้สามารถถูกเรียกผ่านทาง API ได้ดังรูปที่ 4.16



**รูปที่ 4.16** รูปเมนูการตั้งค่า Build triggers ของ Jenkins

เนื่องจากว่าเราจะทำการตั้งค่าไว้ว่า เมื่อ MicroData เกิดการเปลี่ยนแปลงซอร์สโค้ดที่ version control system จะให้ทำการเรียก API ด้วย token ที่กำหนดเพื่อให้เกิดการรัน build job นี้โดยอัตโนมัติ เมื่อซอร์สโค้ดเปลี่ยนแปลง จากรูปเราจะทำการตั้ง token ไว้ว่า MICRODATAAUTOMATED ซึ่งการที่ตั้ง trigger remotely นั้นหากมีการส่งคำร้อง HTTP ไปที่ http://yourdomain/jenkins/job/MicroDataEndToEndAutomatedTest/build?token=MICRODATAAUTOMATED แล้ว build job ของเราก็จะถูกเรียกขึ้นทำงานโดยอัตโนมัติ จากนั้นทำการเลือกตำแหน่งของไฟล์ pom.xml ไปที่โปรเจคของเราและกด save เมื่อกลับมาที่หน้าแรกของ Jenkins จะเห็น job ของเราพร้อมสถานภาพดังรูปที่ 4.17



**รูปที่ 4.17** build job พร้อมสถานภาพ

โดยตอนนี้เราจะได้ build job ของเรามาแล้วโดยสัญลักษณ์วงกลมสีเทาเนื่องจากว่า job นี้ยังไม่เคยถูก build มาก่อน ในขั้นตอนต่อไปเราจะทำการสร้าง feature test runner เพื่อให้ build ของเรารันเพื่อไปรัน feature ให้อีกที และเขียน listener class เพื่อมาทำการสั่งรัน build job ตัวนี้

**4.3.2 การเขียน test runner สำหรับ feature**

เนื่องจาก build script นั้นไม่สามารถสั่งรัน feature file ในโปรเจคได้โดยอัตโนมัติ ดังนั้นเราจึงต้องสร้าง unit test ขึ้นมาเพื่อให้ unit test เป็นตัวสั่งรัน feature file ที่เราต้องการให้รันทุกๆรอบแบบอัตโนมัติอีกทีหนึ่ง โดย Class ของ unit test ที่ใช้สั่งรัน feature จะมีลักษณะดังรูปที่ 4.18



**รูปที่ 4.18** Class feature test runner

จากรูปคือ Class feature test runner ซึ่งจะเห็นว่าเราจะทำการใส่ annotation ชื่อ RunWith พร้อมค่า Cucumber.class เอาไว้ด้านบนของ Class เพื่อเป็นการบอกว่า unit test ตัวนี้มีไว้สำหรับรัน feature file หลังจากนั้นก็จะใส่ annotation CucumberOptions เพื่อทำการตั้งค่าต่างๆเกี่ยวกับ runner ตัวนี้ โดยจะมี attribute ที่เอาไว้กำหนดการตั้งค่าได้แก่ features ซึ่งเป็น attribute ที่เอาไว้บอกว่า จะให้ runner ตัวนี้นั้นรัน feature file ใดบ้าง โดยในที่นี้เราจะกำหนดให้รัน feature หลักๆทั้งหมดยกเว้น feature ที่ทำหน้าที่ในการ load test เนื่องจากว่า เราไม่ได้ต้องการทำ load test ในทุกๆครั้งของการทดสอบ ส่วน attribute ต่อมาคือ glue เพื่อบอก runner ว่าให้ดึง Class ที่เรากำหนดไว้มาด้วยเพื่อใช้ step ต่างๆภายใน class หากเราไม่ทำการ glue class ที่มี step ที่ feature เราใช้ เมื่อมีการรัน step ที่หาตัว glue ไม่ได้ก็จะเกิดการ skip เกิดขึ้น ส่วน attribute tags มีไว้เพื่อบอกว่า จะให้รัน scenario หรือ feature ที่มี tags ใดใส่ไว้ ในที่นี้คือ ~@ignore ซึ่งหมายถึงให้รันทุก scenario ที่ไม่มี tag @ignore ใส่อยู่เนื่องจาก tag @ignore มีไว้สำหรับใส่ให้ feature หรือ scenario ที่เราไม่ต้องการรัน

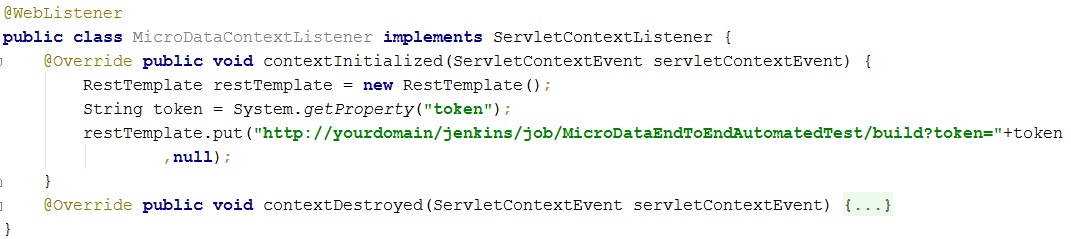
**4.3.3 การเขียน listener สำหรับระบบ MicroData**

ในส่วนของ listener นี้ หลักการก็คือเมื่อระบบถูก deploy เสร็จสิ้น โปรเจคที่เป็น J2EE โปรเจค จะมีการสร้างสิ่งที่เรียกว่า Context เพื่อกำหนดค่าเบื้องต้นต่างๆให้กับโปรแกรม ดังนั้นสิ่งที่เราจะทำก็คือ เราจะทำการสร้าง Class listener ตัวหนึ่งที่คอยให้เหตการณ์การสร้าง Context สำเร็จ ดังนั้นเมื่อ Context ถูกสร้าง listener ตัวนี้ก็จะถูกเรียกขึ้นทำงาน โดยเราจะใช้ช่วงจังหวะนี้ ในการส่งคำร้อง เพื่อไปสั่งให้ build job ใน Jenkins ที่เราเตรียมเอาไว้ก่อนหน้าทำงานเพื่อทำการทดสอบระบบในทันที หลังจากระบบถูก deploy

โดยการสร้าง listener ของเหตุการณ์การสร้าง Context ของ J2EE นั้นจะทำได้โดยการสร้าง Class Java ที่ implements interface และต้อง override เมทอด 2 ตัว คือ

1. contextInitialized เป็นเมทอดที่จะถูกเรียกเมื่อ context ถูกสร้าง
2. contextDestroyed เป็นเมทอดที่ถูกเรียกเมื่อ context ถูกทำลาย

หลังจากนั้นจะทำการลงทะเบียน listener สู่ระบบด้วยการเพิ่ม annotation @WebListener ไว้ที่ Class ของเรา หรือทำการลงทะเบียนใน web.xml ด้วย tag <listener> เหมือนกับการลงทะเบียน servlet ทั่วไปในการทำเว็บแอพพลิเคชั่นก็ได้ หลังจากนั้นเราจะทำการเขียนคำสั่งให้กับเมทอด contextInitialized ให้ทำการส่งคำร้องเพื่อไปสั่งให้ build job ของเราทำงานดังรูปที่ 4.19



**รูปที่ 4.19** Class MicroDataContextListener

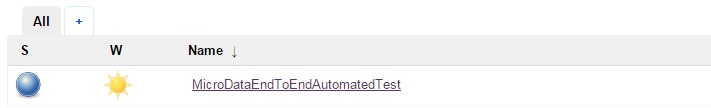
จากรูปจะเห็นว่าเราจะใช้เมทอดของ Class RestTemplate ซึ่งเป็น library ของ Spring framework ในการส่ง request ประเภท PUT ไปที่ url ที่เราได้เตรียมเอาไว้ซึ่งชี้ไปที่ build job ของเรา พร้อมกับ token ที่กำหนดเอาไว้ เพื่อสั่งให้ build job ทำงาน

โดยเมื่อระบบถูก deploy เรียบร้อยก็จะทำการสั่งรัน build job โดยอัตโนมัติ ซึ่งการตรวจสอบจะดูได้จาก console ของ Jenkins สำหรับ build job ของเราจะเห็นว่าใน build history จะมี build เกิดขึ้น ตัวอย่างดังรูปที่ 4.20



**รูปที่ 4.19** ตัวอย่าง build ที่ถูกเรียกจาก context

จะเห็นได้ว่ามี build เกิดขึ้นและเป็นสีฟ้า แสดงว่าทดสอบแล้วระบบทำงานได้ถูกต้องและเราสามารถไปดูได้ที่หน้าแรกว่า ณ ปัจจุบัน build job ของเราอยู่ในสถานภาพใด เนื่องจากในการทำงานจริง build job จะถูกเรียกอยู่เรื่อยๆ ดังรูปที่ 4.20

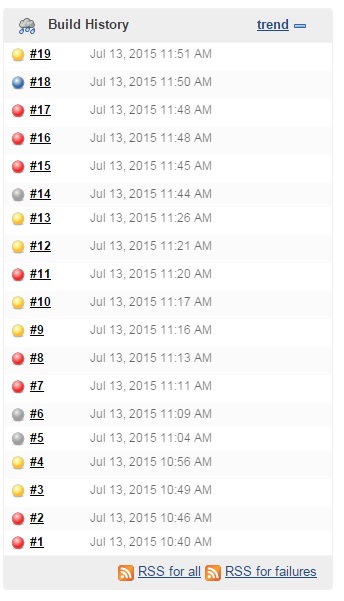


**รูปที่ 4.20** ตัวอย่างสถานะ build ที่ทดสอบถูกต้อง

**4.3.4 รายงานผลการรันงานบน Jenkins**

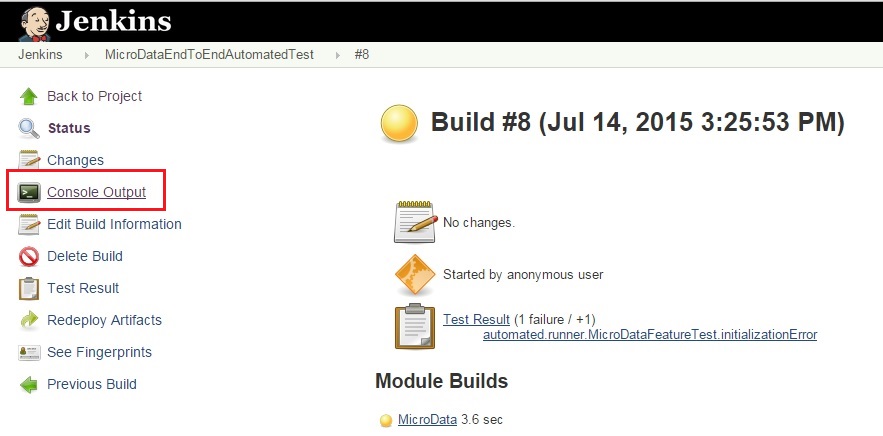
สำหรับในส่วนของผลการรัน build job นั้นสามารถดูได้ 2 ทาง คือ

1. ทาง e-mail โดยเราสามารถตั้งค่าให้ Jenkins ทำการส่ง e-mail ไปหาเราได้ในทุกครั้งที่ build job ทำการรันเสร็จสิ้น โดยจะส่งไปในรูปของรายงาน ที่บอกว่า build สำเร็จ หรือ มีข้อผิดพลาด พร้อมลิงค์เพื่อให้เข้าไปตรวจสอบผลเต็มๆได้ที่หน้าเว็บของ Jenkins
2. ตรวจสอบโดยตรงที่หน้า Jenkins โดยเข้าไปที่เมนูของ build job ของเราเอง เพื่อดูว่า build ไหนที่มีข้อผิดพลาด หรือ build ปัจจุบันเป็นอย่างไรบ้างดังรูปที่ 4.21



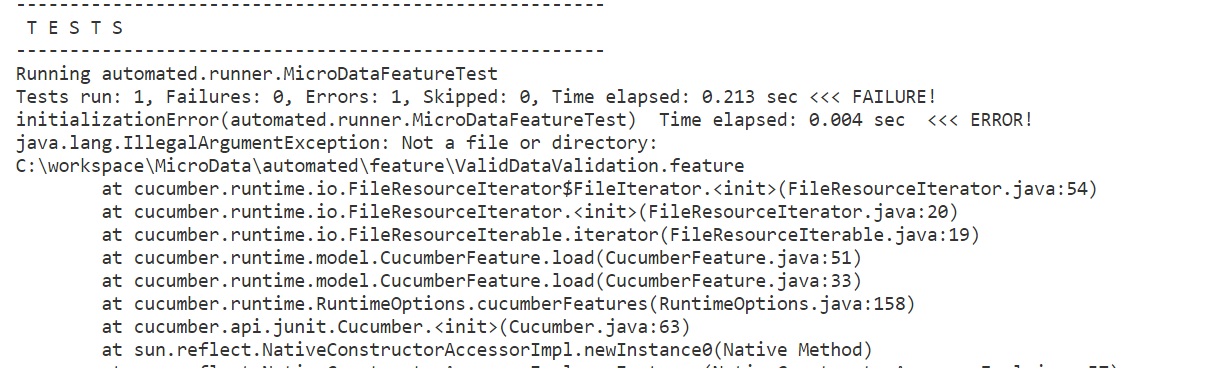
**รูปที่ 4.21** ตัวอย่าง build history ของ Jenkins

โดยในกรณีที่ build fail เราสามารถเข้าไปเพื่อดูข้อมูลของ build นั้นแล้วไปคลิกไปที่เมนู console output ในรูปที่ 4.22



**รูปที่ 4.21** เมนู console output ของ build job

ซึ่งเราสามารถรู้ได้ว่า build ที่ fail เพราะ step หรือ Classใดทำงานผิดพลาด เช่นเดียวกันกับ รายงานของ IntelliJ ในหัวข้อก่อนหน้า หน้า console output จะแสดงดังรูปที่ 4.22



**รูปที่ 4.22** ตัวอย่างรายงานใน console output