RED HAT®
CONSULTING

redhat.

WHITEPAPER

A STRUCTURED DESIGN APPROACH TO BUSINESS AUTOMATION

Advances in software development infrastructure and approaches provide the tools and opportunity to modernize business automation. A design approach that applies best practices from modern software development to business automation can help organizations add rigorous engineering practices while eliminating additional risk.

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com

EXECUTIVE SUMMARY

In many leading companies, IT must provide tools to help the business respond to competition, comply with industry regulations, and engage customers. IT is expected to provide solutions that are high quality, flexible enough for rapid, frequent change, and available at predictable—preferably low—cost.

Custom software development and business automation are two ways to give business stakeholders the solutions they want, when they want them. But the two approaches have different benefits and drawbacks. Software development is an engineering discipline with rigorous, but historically slow, processes. Business automation reduces time to market by letting nontechnical stakeholders codify business logic, resulting in increased risk.

Advances in software development infrastructure and approaches provide the tools and opportunity to modernize business automation. A design approach that applies best practices from modern software development to business automation can help organizations take advantage of this opportunity by adding rigorous engineering practices while eliminating additional risk. Using this approach to business automation, businesses can engage subject matter experts to codify business logic in a way that ensures the higher quality, faster time to market, and lower, predictable cost associated with modern software development.

THE EVOLUTION OF BUSINESS AUTOMATION AND SOFTWARE DEVELOPMENT

IT leaders are always looking for new products or technologies that help them meet business demands. But commercial software often does not let companies quickly innovate or sufficiently differentiate themselves, forcing IT teams to develop custom software.

Historically, software development was slow and complex, requiring effective communication from the start between business experts and developers. A waterfall method was common—in which the business expert would provide requirements to the developer up front, then the developer would design, develop, and test the application. However, the business expert could only assess the application once it was completed, making changes difficult and expensive.

Business automation, also called business process automation, emerged as an alternative to IT-led software development. This method lets business experts quickly and iteratively define and execute the business logic they require. Business logic is expressed as sets of processes, rules, or workflows, rather than as application functions. These rules are executed by an engine in response to specific conditions or events.

For example, the process of onboarding new customers of financial institutions is regulated by governments at all levels to prevent money laundering and other illegal activity, but these regulations change frequently. A compliance specialist at a financial institution can use business automation to define conditions and related rules for new regulations. The specialist can then deploy these new rules into an engine without assistance from software architects, developers, or other IT staff.

Although business automation and custom software development share a goal—to encode business logic in a software system—each was established separately. The resulting isolated, fragmented IT led to higher costs from duplication of servers, operating systems and other software licenses, maintenance and support, and more. And by avoiding the constraints of software development, business automation might have inadvertently increased risk by avoiding rigorous engineering practices. For example, software engineering principles such as maintaining a history of changes for version control, tracking what is tested and ready for production release, and managing configuration—including the life cycles of many components—may have been overlooked in a business automation environment.

In the past few decades, there have been significant changes in software design and development. In many cases, these changes were a response to the same criticism that initially prompted the rise of business automation. Software architecture has progressed from designs based on huge, monolithic applications—slow to build and difficult to change—to architectures that are based on services, with application components that are designed for a single purpose and are easier to create, update, and maintain.

Software development methods have also changed. A traditional waterfall method is characterized by careful planning and distinct phases, making it difficult and complex to repeat parts of the process. This method has been replaced with iterative approaches that aim to quickly get an application working, review with the stakeholder, and make changes with each frequent iteration. These frameworks, such as extreme programming, Scrum, and Scaled Agile Framework, have improved over time in response to technological advances. Today, the practices of continuous integration and continuous delivery (CI/CD) extend a fast, iterative approach to deployment. These practices are built on the idea that code should be integrated multiple times each day and tested throughout development to let software be released at any time.

A MODERN DESIGN APPROACH TO BUSINESS AUTOMATION

Many business automation solutions have missed the opportunity to take advantage of progress in software design and development. An effective design approach for bringing the benefits of modern software development to business automation organizes design concerns into a set of architectural components that each require at least one key implementation decision. An environment built with this approach lets organizations apply existing software development infrastructure to business automation to support both.

The architectural components correspond to the primary activities of an effective business automation pathway:

1. **Model** the business logic and test scenarios that define it by developing application code or defining the basic elements of business automation: rules, events, processes, and workflows.
2. **Version** the model's artifacts, including an auditable history of changes.
3. **Build** a deployable package from a collection of artifacts, run test automation, and maintain a history of the package.

4. **Store** the package in a master repository.
5. **Deploy** and test the package in an execution environment—such as an application server for code or an engine for business automation elements.
6. **Execute** the business logic.

The following sections discuss the architectural components and key implementation decisions. The complexity, capabilities, and cost of an environment depend on the decisions made for each component.

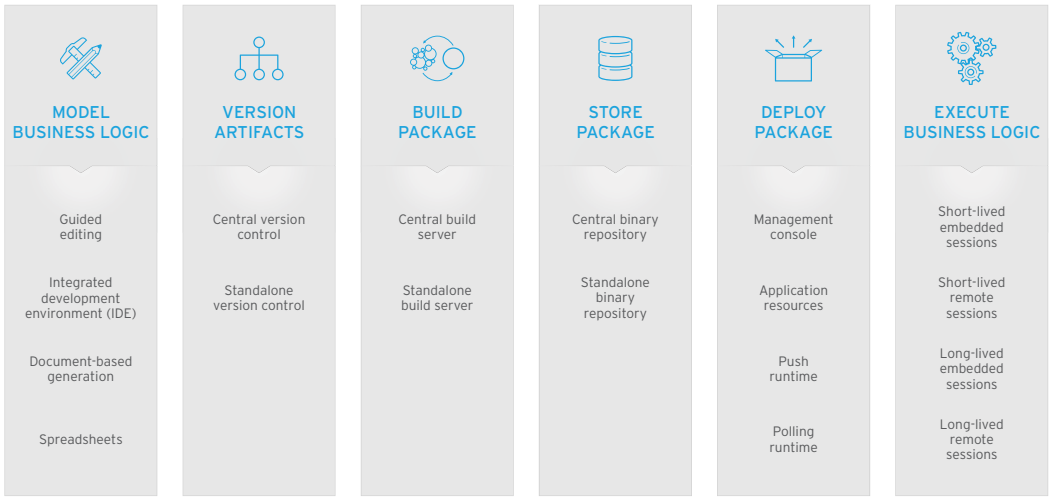

Figure 1. Architectural components and corresponding implementation decisions for business automation

1. MODEL

Modeling is the task of capturing business logic for execution. Users typically model business logic based on their knowledge of the system's expected behavior. Business logic can be modeled with application code, with the elements of business automation, or with a combination of both methods.

With application code, programmers model business logic using a programming language. Many programmers use an integrated development environment (IDE) to write code, but any text editor can be used, as long as it produces standard-compliant code.

With business automation, business subject matter experts model business logic by creating rules, processes, events, and workflows using several methods:

- **Guided editing tool.** A guided editing tool provides an intuitive interface that hides technical details and does not require technical knowledge. With this tool, the user defines test cases to validate accuracy. Guided editing fulfills the original promise of business automation by providing a tool that lets business experts perform modeling without the help of a programmer.

- **Spreadsheet.** Business experts can also use spreadsheets to define rules as decision tables—commonly configured with a repeating pattern across the rows. Business users are usually familiar with spreadsheets and can easily edit each row's values to define the desired logic. Spreadsheets can also be used for process modeling.
- **Document-based generation.** With this approach, a template defines the structure of a rule or process. Business data stored in a document—such as a contract or a promotion offer—is applied to the template, and rules or processes are generated.
- **Integrated development environments.** IDEs are used by programmers to write application code and can be used to create business automation elements. An IDE requires a plug-in to let users create business rules and process diagrams. IDEs provide additional features, such as syntax highlighting, and can offer a familiar working environment for more technically focused users.

When business logic is modeled, business experts can use behavior-driven development (BDD) to provide guidance for the development and eventual testing of the application. With BDD, business experts create short scenarios that describe the desired behavior of the application in an understandable language and constitute the requirements for the application. These scenarios are used by developers—or by business experts, in a business automation approach—to model business logic. In later stages of the development life cycle, these same scenarios are used as the basis for automated tests of the application.

2. VERSION

The version control system is the master repository of business logic artifacts. Whether the business logic is captured as application code or as business rules—and regardless of the tool used—the artifacts that are created must be saved. Versioning, however, goes beyond simply saving the artifact to record all changes to artifacts over time, note the reason for each change, and enable recovery of a specific past version. Versioning can also support multiple branches of an artifact for tracking more than one sequence of changes.

Traditionally, business automation products use databases to save versions of business automation artifacts, but databases lack some of the benefits of a versioning tool. Standard, file-based version control systems—such as Git, Mercurial, or Apache Subversion—are commonly used for application code, and many organizations already use them to create centralized, shared version control. Using an existing version control system frees programmers or business experts from concerns such as security or data retention and backup. However, if a centralized versioning infrastructure is not available or is too complex or costly to integrate, business logic artifacts can be saved in a standalone, dedicated version control system.

3. BUILD

The build system is the primary process for preparing and combining artifacts into packages for eventual release. It is important to efficiently and consistently build the artifacts into a package—ideally with little or no user intervention.

Packages are usually created with a build automation tool such as Apache Maven. The build automation tool is configured to not only run any commands that are needed to obtain artifacts from the version control system, but also to prepare and combine them for eventual release. Builds can be automated to create a new package whenever a new artifact is saved in the version control system.

Automated testing of the new package is completed during this step as part of continuous integration. This testing quickly identifies any integration errors for resolution before additional code is versioned and new packages are built. As a result, a package with integration errors should not continue past this step in the release process.

As with the version control system, many organizations have an existing build automation system. And by using the existing system, users are freed from the responsibility of installing, configuring, and maintaining that system. In addition, packaged software assets often must comply with company policies and external regulations, which are already addressed in the centralized build system.

If the centralized build system does not support Maven or there are other barriers to using the existing system, a separate build system can be used. Creation of a separate build system results in multiple build systems—for example, one per application team or even one for each programmer and business expert.

4. STORE

The package repository is the master source of ready-to-deploy software packages. Software packages are often deployed multiple times to different servers in different datacenters. Deploying stored packages from the repository is more efficient and consistent than rebuilding the package each time it must be deployed—and ensures the exact same package is deployed every time.

As mentioned previously, a centralized repository is often used to address requirements for security, governance, regulatory compliance, data retention, and auditing. And again, it is possible to use a standalone, Maven-compliant repository, though using one results in multiple repositories throughout the organization.

5. DEPLOY

Software packages are deployed from the package repository into an execution environment that depends on the type of package. For example, a standalone, binary executable application will be copied to the server where it will run, while an enterprise Java application might be loaded into an application server. Business automation packages are deployed into the business automation engine—the rules and process execution environment.

Packages can be deployed with little or no user intervention. When initiated by a user, deployment steps are still predefined and previously validated to reduce the potential for error. Deployment rules, such as the priority and schedule for deployment, can also be formalized.

Deployment rules are also used to define required pre-release testing. Before a package is deployed to a production environment, it is typically deployed to a quality assurance (QA) tier and a preproduction tier for testing. These tests are more extensive than the automated testing in the Build step. For example, in QA, a tester will act as the user to try out new features, while in preproduction, load and performance testing is often conducted. Packages are deployed to production only after testing has been completed successfully in these tiers.

The preferred way to deploy packages is by push, where an automated process on one system deploys the package to the execution environment. With a continuous delivery (CD) approach, deployment can occur once the package is updated in the repository or can be configured to follow a schedule. Every execution environment has the latest version of the package according to the deployment rules, eliminating the need for user involvement.

A similar approach uses a management console, typically a web application with a user interface, to deploy packages. The user configures parameters for deploying the package—such as name, version, and target execution environment—and initiates the deployment manually. The management console then pushes the package to the execution environment. The management console push is a user-initiated push, rather than automated.

A pull, or polling, approach can also be used. In this approach, the execution environment scans for and retrieves updated packages from the repository. This approach adds operational complexity by creating a tighter connection between the Build and Deploy steps, because the updated package will be pulled as soon as it is built. For example, if multiple execution environments—such as development, test, and production environments—use the same repository, all environments will pull the latest build. While there are several ways to accomplish this goal, push mechanisms are preferred.

For business automation artifacts, the application resources approach is another deployment option. In this approach, artifacts are bundled with the execution engine into the application. This is a simple, unified approach, but it creates a monolithic application. Deployment is similar to a traditional application, rather than a business automation package. Ideally, this approach is paired with a push or management console approach.

6. EXECUTE

The execution environment generally includes all of the servers, storage, operating systems, networks, and other systems that are required to execute business logic. For the purposes of this paper, only the business automation component is considered.

The implementation considerations for the execution environment are unlike those for the other architectural components in the approach described in this paper. The Model step depends on user skill and personal preference, while Version, Build, and Store only require a decision on the location of the repository—and this decision can be changed at any time. Considerations for the Deploy step affect how the packages are placed in the execution environment but not how they are actually executed.

The execution environment affects how business logic executes, addressing performance, scalability, and reliability through technical design. Design decisions in this environment can be key to customer satisfaction—and determine success or failure.

Determining whether the business automation engine is distributed as part of the application or established as a centralized shared service is a critical design decision. With embedded execution, the engine is part of an application and is dedicated to serving that application. The application and engine run together in the same Java™ virtual machine to produce the highest performance. But this type of execution is less cost-effective, because additional computing power is required anywhere the engine runs.

Establishing a centralized shared service, however, separates application logic from the business automation engine and the rules and process deployed to it. A single, consistent engine is available throughout the organization for any applications. As a result, applications need less computing power, and many applications can be supported by scaling this shared engine as needed. Using this approach requires distributed programming techniques but provides engine access using a variety of protocols, making the engine available to applications that are not Java-based.

Another key decision relates to how applications interact with the engine. Some applications execute a rule and return a result in a single interaction. For these short-lived sessions, the engine does not store any application data after the rule executes. Typically, the application manages data in an independent persistent store and passes it to the engine as needed.

Other applications execute a business process that can span days, weeks, or longer. In these interactions, the application creates a long-lived session with the engine, and the engine is responsible for persisting application data to use in subsequent steps. In this case, the data can also be accessed by other applications.

Session lifespan and state management can significantly affect the execution environment's performance and scalability. Short-lived sessions can be scaled horizontally without adding complexity, while long-lived sessions can be scaled vertically or horizontally—but require a well-designed database. In general, because of their resource requirements, long-lived sessions are used only when necessary, such as for business process management.

CONCLUSION

Business automation emerged as a result of the inability of software development to meet modern business needs. But to meet those needs, business automation created increased risk and other challenges. In the last 20 years, the pace of innovation has accelerated beyond what traditional business automation solutions can support. The economy is now defined by digital disruption of mature markets and a rapidly evolving, increasingly complex regulatory environment.

Industry research, such as Puppet's 2015 State of DevOps Report,[1] demonstrates that high-performing software organizations are significantly more prepared to compete in this volatile landscape. Therefore, business automation solutions must incorporate modern infrastructure and approaches for software development and deployment.

Red Hat® Consulting's Business Automation practice uses the unified approach presented in this paper to help organizations achieve the value of business automation by taking advantage of modern software development techniques, rather than circumventing them.

To learn more about Red Hat Consulting's Business Automation practice, read the datasheet at redhat.com/en/resources/red-hat-consulting-discovery-session-business-automation.

1 *https://puppet.com/resources/white-paper/2015-state-of-devops-report*

ABOUT RED HAT

Red Hat is the world's leading provider of open source software solutions, using a community-powered approach to provide reliable and high-performing cloud, Linux, middleware, storage, and virtualization technologies. Red Hat also offers award-winning support, training, and consulting services. As a connective hub in a global network of enterprises, partners, and open source communities, Red Hat helps create relevant, innovative technologies that liberate resources for growth and prepare customers for the future of IT.

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com
#INC0410962_v1_201606_KVM

NORTH AMERICA
1 888 REDHAT1

EUROPE, MIDDLE EAST, AND AFRICA
00800 7334 2835
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
info-latam@redhat.com