Developer Experience (DevEx)

Developer experience refers to how easy or difficult it is for a developer to perform essential tasks needed to implement a change. A positive developer experience would mean these tasks are relatively easy for the team (see measures below).

The essential tasks are identified below.

- Build - Verify that changes are free of syntax error and compile.
- Test - Verify that all automated tests pass.
- Start - Launch end-to-end to simulate execution in a deployed environment.
- Debug - Attach debugger to started solution, set breakpoints, step through code, and inspect variables.

If effort is invested to make these activities as easy as possible, **the returns on that effort will increase the longer the project runs, and the larger the team is**.

Defining End-to-End

This document makes several references to running a solution end-to-end (aka E2E). End-to-end for the purposes of this document is scoped to the software that is owned, built, and shipped by the team. Systems owned by other teams or third-party vendors is not within the E2E scope for the purposes of this document.

Goals

- Maximize the amount of time engineers spend on writing code that fulfills story acceptance and done-done criteria.
- Minimize the amount of time spent manual setup and configuration of tooling
- Minimize regressions and new defects by making end-to-end testing easy

Impact

Developer experience can have a significant impact on the efficiency of the day-to-day execution of the team. A positive experience can pay dividends throughout the lifetime of the project; especially as new developers join the team.

- Increased Velocity - Team spends less time on non-value-add activities such as dev/local environment setup, waiting on remote environments to test, and rework (fixing defects).
- Improved Quality - When it's easy to debug and test, developers will do more of it. This will translate to fewer defects being introduced.
- Easier Onboarding & Adoption - When dev essential tasks are automated, there is less documentation to write and, subsequently, less to read to get started!

Most importantly, the customer will continue to accrue these benefits long after the code-with engagement.

Measures

Time to First E2E Result (aka F5 Contract)

Assuming a laptop/pc that has never run the solution, how long does it take to set up and run the whole system end-to-end and see a result.

Time To First Commit

How long does it take to make a change that can be verified/tested locally. A locally verified/tested change is one that passes test cases without introducing regression or breaking changes.

Participation

Providing a positive developer experience is a team effort. However, certain members can take ownership of different areas to help hold the entire team accountable.

Dev Lead - Set the Bar

The following are examples of how the Dev Lead might set the bar for dev experience

- Determines development environment (suggested IDE, hosting, etc)
- Determines source control environment and number of repos required
- Given development environment and repo structure, sets expectations for team to meet in terms of steps to perform the essential dev tasks
- Nominates the DevEx Champion

IDE choice is NOT intended to mandate that all team members must use the same IDE. However, this choice will direct where tight-integration investment

will be prioritized. For example, if Visual Studio Code is the **suggested** IDE then, the team would focus on integrating VS code tasks and launch configurations over similar integrations for other IDEs. Team members should still feel free to use their preferred IDE as long as it does not negatively impact the team.

2 DevEx Champion - Identify Iterative Improvements

The DevEx champion takes ownership in holding the team accountable for providing a positive developer experience. The following outline responsibilities for the DevEx champion.

- Actively seek opportunities for improving the solution developer experience
- Work with the Dev Lead to iteratively improve team expectations for developer experience
- Curate a backlog actionable stories that identify areas for improvement and prioritize with respect to project delivery goals by engaging directly with the Product Owner and Customer.
- Serve as subject-matter expert for the rest of the team. Help the team determine how to implement DevEx expectations and identify deviations.

3 Team Members - Assert Expectations

The team members of the team can also help hold each other accountable for providing a positive developer experience. The following are examples of areas team members can help identify where the team's DevEx expectations are not being met.

- Pull requests. Try the changes locally to see if they are adhering to the

team's DevEx expectations.

- Design Reviews. Look for proposals that may negatively affect the solution's DevEx. These might include
  - Introduction of new tech whose testability is limited to manual steps in a deployed environment.
  - Addition of new repository

New Team Members - Identify Iterative Improvements

New team members are uniquely positioned to identify instances of undocumented Collective Wisdom. The following outlines responsibilities of new team members as it relates to DevEx:

- If you come across missing, incomplete or incorrect documentation while onboarding, you should record the issue as a new defect(s) and assign it to the product owner to triage.
- If no onboarding documentation exists, note the steps you took in a new user story. Assign the new story to the product owner to triage.

Facilitation Guidance

The following outline examples of several strategies that can be adopted to promote a positive developer experience. It is expected that each team should define what a positive dev experience means within the context of their project. Additionally, refine that over time via feedback mechanisms such as sprint and project retrospectives.

Establish Hotkeys

Assign hotkeys to each of the essential tasks.

Task	Windows
Build	CTRL+SHIFT+B
Test	CTRL+R,T
Start With Debugging	F5

The F5 Contract

The F5 contract aims for the ability to run the end-to-end solution with the following steps.

1. Clone - git clone [`my-repo-url-here`]
2. Configure - set any configuration values that need to be unique to the individual (i.e. update a .env file)
3. Press F5 - launch the solution with debugging attached.

Most IDEs have some form of a task runner that can be used to automate the build, execute, and attach steps. Try to leverage these such that the steps can all be run with as few manual steps as possible.

DevEx Champion Actively Seek Improvements

The DevEx champion should actively seek areas where the team has opportunity to improve. For example, do they need to deploy their changes to an environment off their laptop before they can validate if what they did

worked. Rather than debugging locally, do they have to do this repetitively to get to a working solution? Does this take several minutes each iteration? Does this block other developers due to the contention on the environment?

The following are ceremonies that the DevEx champion can use to find potential opportunities

- Retrospectives. Is feedback being raised that relates to the essential tasks being difficult or unwieldy?
- Standup Blockers. Are individuals getting blocked or stumbling on the essential tasks?

As opportunities are identified, the DevEx champion can translate these into actionable stories for the product backlog.

Make Tasks Cross Platform

For essential tasks being standardized during the engagement, ensure that different platforms are accounted for. Team members may have different operating systems and ensuring the tasks are cross-platform will provide an additional opportunity to improve the experience.

- See the making tasks cross platform recipe for guidance on how tasks can be configured to include different platforms.

Create an Onboarding Guide

When welcoming new team members to the engagement, there are many areas for them to get adjusted to and bring them up to speed including codebase, coding standards, team agreements, and team culture. By adopting a strong onboarding practice such as an onboarding guide in a centralized location that explains the scope of the project, processes, setup

details, and software required, new members can have all the necessary resources for them to be efficient, successful and a valuable team member from the start.

See the onboarding guide recipe for guidance on what an onboarding guide may look like.

Standardize Essential Tasks

Apply a common strategy across solution components for performing the essential tasks

- Standardize the configuration for solution components
- Standardize the way tests are run for each component
- Standardize the way each component is started and stopped locally
- Standardize how to document the essential tasks for each component

This standardization will enable the team to more easily automate these tasks across all components at the solution level. See Solution-level Essential Tasks below.

Solution-level Essential Tasks

Automate the ability to execute each essential task across all solution components. An example would be mapping the build action in the IDE to run the build task for each component in the solution. More importantly, configure the IDE start action to start all components within the solution. This will provide significant efficiency for the engineering team when dealing with multi-component solutions.

When this is not implemented, the engineers must repeat each of the

essential tasks manually for each component in the solution. In this situation, the number of steps required to perform each essential task is multiplied by the number of components in the system

[Configuration steps + Build steps + Start/Debug steps + Stop steps + Run test steps + Documenting all of the above] * [many solution components] = TOO MANY STEPS

VS.

[Configuration steps + Build steps + Start/Debug steps + Stop steps + Run test steps + Documenting all of the above] * [1 solution] = MINIMUM NUMBER OF STEPS

Observability

Observability alleviates unforeseen challenges for the developer in a complex distributed system. It identifies project bottlenecks quicker and with more precision, enhancing performance as the developer seeks to deploy code changes. Adding observability improves the experience when identifying and resolving bugs or broken code. This results in fewer or less severe current and future production failures.

There are many observability strategies a developer can use alongside best engineering practices. These resources improve the DevEx by ensuring a shared view of the complex system throughout the entire lifecycle. Observability in code via logging, exception handling and exposing of relevant application metrics for example, promotes the consistent visibility of real time performance. The observability pillars, logging, metrics, and tracing, detail when to enable each of the three specific types of observability.

Minimize the Number of Repositories

Splitting a solution across multiple repositories can negatively impact the above measures. This can also negatively impact other areas such as Pull Requests, Automated Testing, Continuous Integration, and Continuous Delivery. Similar to the IDE instances, the negative impact is multiplied by the number of repositories.

[Clone steps + Branching steps + Commit steps + CI steps + Pull Request reviews & merges] * [many source code repositories] = TOO MANY STEPS

VS.

[Clone steps + Branching steps + Commit steps + CI steps + Pull Request reviews & merges] * [1 source code repository] = MINIMUM NUMBER OF STEPS

Atomic Pull Requests

When the solution is encapsulated within a single repository, it also allows pull requests to represent a change across multiple layers. This is especially helpful when a change requires changes to a shared contract between multiple components. For example, a story requires that an api endpoint is changed. With this strategy the api and web client could be updated with the same pull request. This avoids the main branch being broken temporarily while waiting on dependent pull requests to merge.

Minimize Remote Dependencies for Local Development

The fewer dependencies on components that cannot run a developer's machine translate to fewer steps required to get started. Therefore, fewer dependencies will positively impact the measures above.

The following strategies can be used to reduce these dependencies

Use an Emulator

If available, emulators are implementations of technologies that are typically only available in cloud environments. A good example is the CosmosDB emulator.

Use DI + Toggle to Mock Remote Dependencies

When the solution depends on a technology that cannot be run on a developer's machine, the setup and testing of that solution can be challenging. One strategy that can be employed is to create the ability to swap that dependency for one that can run locally.

Abstract the layer that has the remote dependency behind an interface owned by the solution (not the remote dependency). Create an implementation of that interface using a technology that can be run locally. Create a factory that decides which instance to use. This decision could be based on environment configuration (i.e. the toggle). Then, the original class that depends on the remote tech instead should depend on the factory to provide which instance to use.

Much of this strategy can be simplified with proper dependency injection technique and/or framework.

See example below that swaps Azure Service Bus implementation for RabbitMQ which can be run locally.

```
interface IPublisher {
    send(message: string): void
}
class RabbitMQPublisher implements IPublisher {
    send(message: string) {
        //todo: send the message via RabbitMQ
    }
}
class AzureServiceBusPublisher implements IPublisher {
    send(message: string) {
        //todo: send the message via Azure Service Bus
```

```
    }
}
interface IPublisherFactory{
    create(): IPublisher
}
class PublisherFactory{
    create(): IPublisher {
        // use env var value to determine which instance should
be used
        if(process.env.UseAsb){
            return new AzureServiceBusPublisher();
        }
        else{
            return new RabbitMqPublisher();
        }
    }
}
class MyService {
    //inject the factory
    constructor(private readonly publisherFactory:
IPublisherFactory){
    }
    sendAMessage(message: string): void{
        //use the factory to determine which instance to use
        const publisher: IPublisher =
this.publisherFactory.create();
        publisher.send(message);
    }
}
```

The recipes section has a more complete discussion on DI as part of a high productivity inner dev loop

---

Last update: August 22, 2024