**CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY USING DevOps**

**Exercise 1:**

**Get an understanding of the stages in software development lifecycle, the process models, values and principles of agility and the need for agile software development. This will enable you to work in projects following an agile approach to software development.**

**Defining SDLC**

A systematic approach that generates a structure for the developer to design, create and deliver high-quality software based on customer requirements and needs. The primary goal of the SDLC process is to produce cost-efficient and high-quality products. The process comprises a detailed plan that describes how to develop, maintain, and replace the software.

**The 7 Phases Of SDLC (Software Development Life Cycle)**

**Stage 1: Project Planning**

The first stage of SDLC is all about “What do we want?” Project planning is a vital role in the software delivery lifecycle since this is the part where the team estimates the cost and defines the requirements of the new software.

**Stage 2: Gathering Requirements & Analysis**

The second step of SDLC is gathering maximum information from the client requirements for the product. Discuss each detail and specification of the product with the customer. The development team will then analyze the requirements keeping the design and code of the software in mind. Further, investigating the validity and possibility of incorporating these requirements into the software system. The main goal of this stage is that everyone understands even the minute detail of the requirement. Hardware, operating systems, programming, and [security](https://www.betsol.com/security-services/) are to name the few requirements.

**Stage 3: Design**

In the design phase (3rd step of SDLC), the program developer scrutinizes whether the prepared software suffices all the requirements of the end-user. Additionally, if the project is feasible for the customer technologically, practically, and financially. Once the developer decides on the best design approach, he then selects the program languages like Oracle, [Java](https://www.betsol.com/blog/java-memory-management-for-java-virtual-machine-jvm/), etc., that will suit the software.

Once the design specification is prepared, all the stakeholders will review this plan and provide their feedback and suggestions. It is absolutely mandatory to collect and incorporate stakeholder’s input in the document, as a small mistake can lead to cost overrun.

**Stage 4: Coding or Implementation**

Time to code! It means translating the design to a computer-legible language. In this fourth stage of SDLC, the tasks are divided into modules or units and assigned to various developers. The developers will then start building the entire system by writing code using the programming languages they chose. This stage is considered to be one of the longest in SDLC. The developers need certain predefined coding guidelines, and programming tools like interpreters, compilers, debugger to implement the code.

The developers can show the work done to the business analysts in case if any modifications or enhancements required.

**Stage 5: Testing**

Once the developers build the software, then it is deployed in the testing environment. Then the testing team tests the functionality of the entire system. In this fifth phase of SDLC, the testing is done to ensure that the entire application works according to the customer requirements.

After testing, the [QA and testing](https://www.betsol.com/software-development-and-testing/) team might find some bugs or defects and communicate the same with the developers. The development team then fixes the bugs and send it to QA for a re-test. This process goes on until the software is stable, bug-free and working according to the business requirements of that system.

**Stage 6: Deployment**

The sixth phase of SDLC: Once the testing is done, and the product is [ready for deployment](https://www.betsol.com/blog/how-to-make-software-deployments-easier/), it is released for customers to use. The size of the project determines the complexity of the deployment. The users are then provided with the training or documentation that will help them to operate the software.  Again, a small round of testing is performed on production to ensure environmental issues or any impact of the new release.

**Stage 7: Maintenance**

The actual problem starts when the customer actually starts using the developed system and those needs to be solved from time to time. Maintenance is the seventh phase of SDLC where the developed product is taken care of. According to the changing user end environment or technology, the software is updated timely.

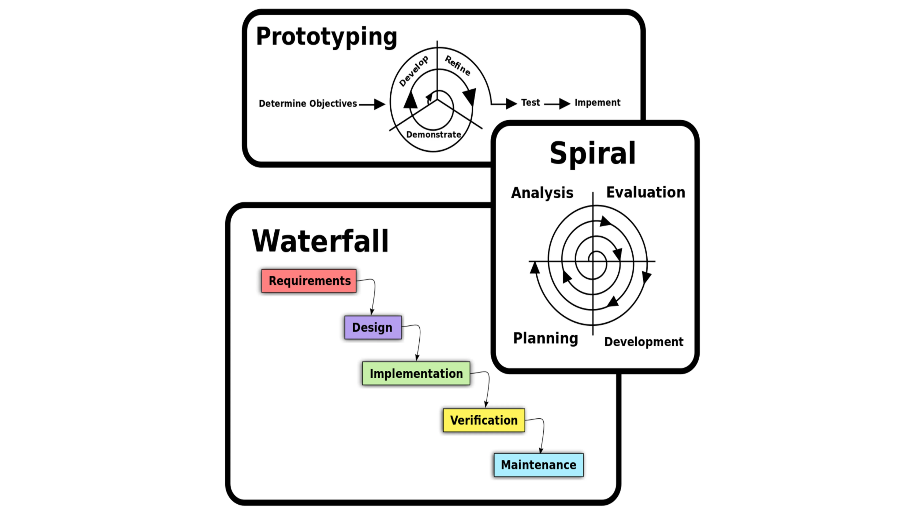
## Software process model

A software process model is an abstraction of the software development process. The models specify the stages and order of a process. So, think of this as a representation of the **order of activities** of the process and the **sequence** in which they are performed.

**A model will define the following:**

* The tasks to be performed
* The input and output of each task
* The pre and post-conditions for each task
* The flow and sequence of each task

The goal of a software process model is to provide guidance for controlling and coordinating the tasks to achieve the end product and objectives as effectively as possible.



There are many kinds of process models for meeting different requirements. We refer to these as **SDLC models** (Software Development Life Cycle models). The most popular and important SDLC models are as follows:

* Waterfall model
* V model
* Incremental model
* RAD model
* Agile model
* Iterative model
* Prototype model
* Spiral model

### Project requirements

Before you choose a model, take some time to go through the project requirements and clarify them alongside your organization’s or team’s expectations. Will the user need to specify requirements in detail after each iterative session? Will the requirements change during the development process?

### Project size

Consider the size of the project you will be working on. Larger projects mean bigger teams, so you’ll need more extensive and elaborate project management plans.

### Project complexity

Complex projects may not have clear requirements. The requirements may change often, and the cost of delay is high. Ask yourself if the project requires constant monitoring or feedback from the client.

### Cost of delay

Is the project highly time-bound with a huge cost of delay, or are the timelines flexible?

### Customer involvement

Do you need to consult the customers during the process? Does the user need to participate in all phases?

### Familiarity with technology

This involves the developers’ knowledge and experience with the project domain, software tools, language, and methods needed for development.

### Project resources

This involves the amount and availability of funds, staff, and other resources.

## Types of software process models

As we mentioned before, there are multiple kinds of software process models that each meet different requirements.

### Waterfall Model

The waterfall model is a **sequential, plan driven-process** where you must plan and schedule all your activities before starting the project. Each activity in the waterfall model is represented as a separate phase arranged in linear order.

It has the following phases:

* Requirements
* Design
* Implementation
* Testing
* Deployment
* Maintenance

Each of these phases produces one or more documents that need to be approved before the next phase begins. However, in practice, these phases are very likely to overlap and may feed information to one another.

The software process **isn’t linear**, so the documents produced may need to be modified to reflect changes.

The waterfall model is easy to understand and follow. It doesn’t require a lot of customer involvement after the specification is done. Since it’s inflexible, it can’t adapt to changes. There is no way to see or try the software until the last phase.

The waterfall model has a rigid structure, so it should be used in cases where the requirements are understood completely and unlikely to radically change.

### V Model

The V model (Verification and Validation model) is an extension of the waterfall model. All the requirements are gathered at the start and cannot be changed. You have a corresponding testing activity for each stage. For every phase in the development cycle, there is an **associated testing phase.**

The corresponding testing phase of the development phase is planned in parallel, as you can see above.

The V model is highly disciplined, easy to understand, and makes project management easier. But it isn’t good for complex projects or projects that have unclear or changing requirements. This makes the V model a good choice for software where downtimes and failures are unacceptable.

### Incremental Model

The incremental model divides the system’s functionality into **small increments** that are delivered one after the other in quick succession. The most important functionality is implemented in the initial increments.

The subsequent increments expand on the previous ones until everything has been updated and implemented.

Incremental development is based on developing an initial implementation, exposing it to user feedback, and evolving it through new versions. The process’ activities are interwoven by feedback.

Each iteration passes through the requirements, design, coding, and testing stages.

The incremental model lets stakeholders and developers see results with the first increment. If the stakeholders don’t like anything, everyone finds out a lot sooner. It is efficient as the developers only focus on what is important and bugs are fixed as they arise, but you need a **clear and complete definition** of the whole system before you start.

The incremental model is great for projects that have loosely coupled parts and projects with complete and clear requirements.

### Iterative Model

The iterative development model develops a system by **building small portions** of all the features. This helps to meet the initial scope quickly and release it for feedback.

In the iterative model, you start off by implementing a small set of software requirements. These are then **enhanced iteratively** in the evolving versions until the system is completed. This process model starts with part of the software, which is then implemented and reviewed to identify further requirements.

Like the incremental model, the iterative model allows you to see the results at the early stages of development. This makes it easy to identify and **fix any functional or design flaws**. It also makes it easier to manage risk and change requirements.

The deadline and budget may change throughout the development process, especially for large complex projects. The iterative model is a good choice for large software that can be **easily broken down into modules**.

### RAD Model

The Rapid Application Development (RAD model) is based on iterative development and prototyping with **little planning involved**. You develop functional modules in parallel for faster product delivery. It involves the following phases:

1. Business modeling
2. Data modeling
3. Process modeling
4. Application generation
5. Testing and turnover

The RAD concept focuses on gathering requirements using focus groups and workshops, reusing software components, and informal communication.

The RAD model accommodates changing requirements, reduces development time, and increases the reusability of components. But it can be complex to manage. Therefore, the RAD model is great for systems that need to be produced in a short time and have known requirements.

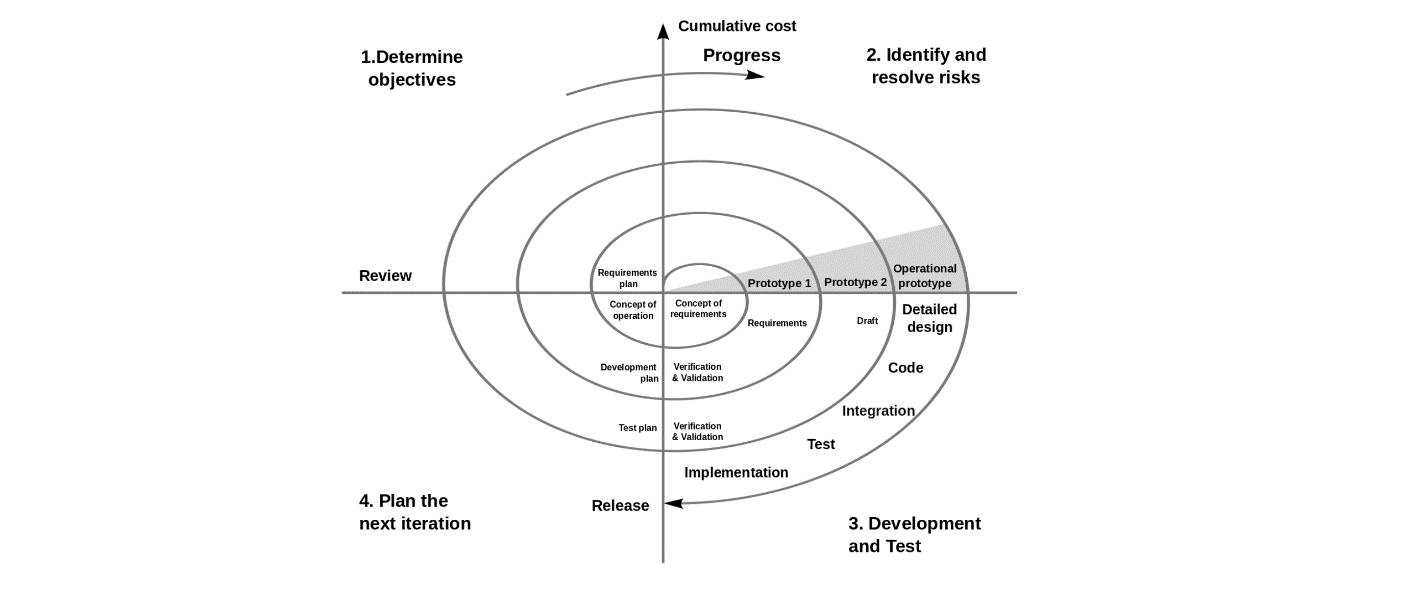
### Spiral Model

The spiral model is a risk driven iterative software process model. The spiral model delivers projects in loops. Unlike other process models, its steps aren’t activities but **phases** for addressing whatever problem has the greatest risk of causing a failure.

It was designed to include the best features from the waterfall and introduces risk-assessment.

You have the following phases for each cycle:

1. Address the highest-risk problem and determine the objective and alternate solutions
2. Evaluate the alternatives and identify the risks involved and possible solutions
3. Develop a solution and verify if it’s acceptable
4. Plan for the next cycle



You develop the concept in the first few cycles, and then it evolves into an implementation. Though this model is great for managing uncertainty, it can be difficult to have stable documentation. The spiral model can be used for projects with **unclear needs** or projects still in research and development.

### Agile model

The agile process model encourages **continuous iterations of development** and testing. Each incremental part is developed over an iteration, and each iteration is designed to be small and manageable so it can be completed within a few weeks.

Each iteration focuses on implementing a small set of features completely. It involves customers in the development process and minimizes documentation by using informal communication.

Agile development considers the following:

* Requirements are assumed to change
* The system evolves over a series of short iterations
* Customers are involved during each iteration
* Documentation is done only when needed

Though agile provides a very realistic approach to software development, it isn’t great for complex projects. It can also present challenges during transfers as there is very little documentation. Agile is great for projects with **changing requirements**.

Some commonly used agile methodologies include:

* **Scrum:** One of the most popular agile models, Scrum consists of iterations called sprints. Each sprint is between 2 to 4 weeks long and is preceded by planning. You cannot make changes after the sprint activities have been defined.
* **Extreme Programming (XP):** With Extreme Programming, an iteration can last between 1 to 2 weeks. XP uses pair programming, continuous integration, test-driven development and test automation, small releases, and simple software design.
* **Kanban:** Kanban focuses on visualizations, and if any iterations are used they are kept very short. You use the Kanban Board that has a clear representation of all project activities and their numbers, responsible people, and progress.

**This will enable you to work in projects following an agile approach to software development.**

# **Jira and Agile Projects**

*Atlassian’s Jira has emerged as the tool of choice for managing agile projects in the enterprise. This class teaches you to align agile practices, roles, and ceremonies with the Jira environment. Return to work ready to use Jira to manage technology projects, IT service delivery, and agile software development.*

**Use Jira for executing everyday agile project work.**

This fast-paced, hands-on workshop teaches you how to manage agile projects, products, and development work using Atlassian Jira as your primary collaboration platform. In Jira, you have a tool designed to enable agile practices and track agile projects across a wide range of scenarios. Features such as issue tracking, search, reporting, and customization allow for seamless integration of agile projects in one project management environment. No other agile project management tool offers the depth of agile context and comprehensive agile framework that Atlassian has made available with JIRA.

**Exercise 2:**

**Get a working knowledge of using extreme automation through XP programming practices of test first development, refactoring and automating test case writing**

## What is Extreme Programming?

XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software.

e**X**treme **P**rogramming (XP) was conceived and developed to address the specific needs of software development by small teams in the face of vague and changing requirements.

Extreme Programming is one of the Agile software development methodologies. It provides values and principles to guide the team behavior. The team is expected to self-organize. Extreme Programming provides specific core practices where −

* Each practice is simple and self-complete.
* Combination of practices produces more complex and emergent behavior.

### Embrace Change

A key assumption of Extreme Programming is that the cost of changing a program can be held mostly constant over time.

This can be achieved with −

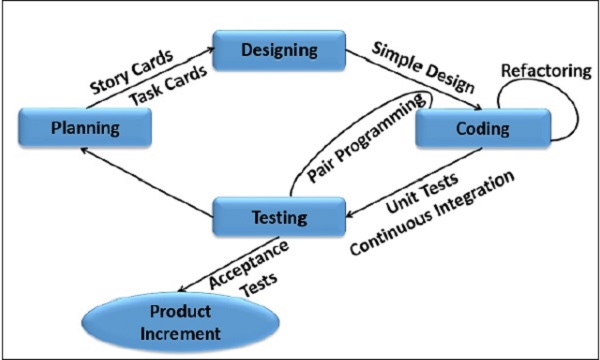
* Emphasis on continuous feedback from the customer
* Short iterations
* Design and redesign
* Coding and testing frequently
* Eliminating defects early, thus reducing costs
* Keeping the customer involved throughout the development
* Delivering working product to the customer

## Extreme Programming in a Nutshell

Extreme Programming involves −

* Writing unit tests before programming and keeping all of the tests running at all times. The unit tests are automated and eliminates defects early, thus reducing the costs.
* Starting with a simple design just enough to code the features at hand and redesigning when required.
* Programming in pairs (called pair programming), with two programmers at one screen, taking turns to use the keyboard. While one of them is at the keyboard, the other constantly reviews and provides inputs.
* Integrating and testing the whole system several times a day.
* Putting a minimal working system into the production quickly and upgrading it whenever required.
* Keeping the customer involved all the time and obtaining constant feedback.

Iterating facilitates the accommodating changes as the software evolves with the changing requirements.



## Why is it called “Extreme?”

Extreme Programming takes the effective principles and practices to extreme levels.

* Code reviews are effective as the code is reviewed all the time.
* Testing is effective as there is continuous regression and testing.
* Design is effective as everybody needs to do refactoring daily.
* Integration testing is important as integrate and test several times a day.
* Short iterations are effective as the planning game for release planning and iteration planning.

## Effective Principles and Practices Success in Industry

The success of projects, which follow Extreme Programming practices, is due to −

* Rapid development.
* Immediate responsiveness to the customer’s changing requirements.
* Focus on low defect rates.
* System returning constant and consistent value to the customer.
* High customer satisfaction.
* Reduced costs.
* Team cohesion and employee satisfaction.

## Extreme Programming Advantages

Extreme Programming solves the following problems often faced in the software development projects −

* **Slipped schedules** − and achievable development cycles ensure timely deliveries.
* **Cancelled projects** − Focus on continuous customer involvement ensures transparency with the customer and immediate resolution of any issues.
* **Costs incurred in changes** − Extensive and ongoing testing makes sure the changes do not break the existing functionality. A running working system always ensures sufficient time for accommodating changes such that the current operations are not affected.
* **Production and post-delivery defects: Emphasis is on** − the unit tests to detect and fix the defects early.
* **Misunderstanding the business and/or domain** − Making the customer a part of the team ensures constant communication and clarifications.
* **Business changes** − Changes are considered to be inevitable and are accommodated at any point of time.
* **Staff turnover** − Intensive team collaboration ensures enthusiasm and good will. Cohesion of multi-disciplines fosters the team spirit.

XP sets out to lower the cost of change by introducing basic values, principles and practices. By applying XP, a system development project should be more flexible with respect to changes.

## Extreme Programming Values

Extreme Programming (XP) is based on the five values −

* Communication
* Simplicity
* Feedback
* Courage
* Respect

### Communication

Communication plays a major role in the success of a project. Problems with projects often arise due to lack of communication. Many circumstances may lead to the breakdown in communication. Some of the common problems are −

* A developer may not tell someone else about a critical change in the design.
* A developer may not ask the customer the right questions, and so a critical domain decision is blown.
* A manager may not ask a developer the right question, and project progress is misreported.
* A developer may ignore something important conveyed by the customer.

Extreme Programming emphasizes continuous and constant communication among the team members, managers and the customer. The Extreme Programming practices, such as unit testing, pair programming, simple designs, common metaphors, collective ownership and customer feedback focus on the value of communication.

XP employs a coach whose job is to notice when the people are not communicating and reintroduce them. Face-to-Face communication is preferred and is achieved with pair programming and a customer representative is always onsite.

### Simplicity

Extreme Programming believes in ‘it is better to do a simple thing today and pay a little more tomorrow to change it’ than ‘to do a more complicated thing today that may never be used anyway’.

* Do what is needed and asked for, but no more.
  + ''Do the simplest thing that could possibly work'' The DTSTTCPW principle.
  + Implement a new capability in the simplest possible way. Also known as the KISS principle ‘Keep It Simple, Stupid!’.
  + A coach may say DTSTTCPW when he sees an Extreme Programming developer doing something needlessly complicated.
  + Refactor the system to be the simplest possible code with the current feature set. This will maximize the value created for the investment made till date.
* Take small simple steps to your goal and mitigate failures as they happen.
* Create something that you are proud of and maintain it for a long term for reasonable costs.
* Never implement a feature you do not need now i.e. the ‘You Aren’t Going to Need It’ (YAGNI) principle.

**Communication and Simplicity support each other.**

The more you communicate the clearer you can see exactly what needs to be done, and you gain more confidence about what really need not be done.

The simpler your system is, the less you have to communicate about the fewer developers that you require. This leads to better communication.

### Feedback

Every iteration commitment is taken seriously by delivering a working software. The software is delivered early to the customer and a feedback is taken so that necessary changes can be made if needed. Concrete feedback about the current state of the system is priceless. The value of the feedback is a continuously running system that delivers information about itself in a reliable way.

In Extreme Programming, feedback is ensured at all levels at different time scales −

* Customers tell the developers what features they are interested in so that the developers can focus only on those features.
* Unit tests tell the developers the status of the system.
* The system and the code provides feedback on the state of development to the managers, stakeholders and the customers.
* Frequent releases enable the customer to perform acceptance tests and provide feedback and developers to work based on that feedback.
* When the customers write new features/user stories, the developers estimate the time required to deliver the changes, to set the expectations with the customer and managers.

### Courage

Extreme Programming provides courage to the developers in the following way −

* To focus on only what is required
* To communicate and accept feedback
* To tell the truth about progress and estimates
* To refactor the code
* To adapt to changes whenever they happen
* To throw the code away (prototypes)

This is possible as no one is working alone and the coach guides the team continuously.

### Respect

Respect is a deep value, one that lies below the surface of the other four values. In Extreme Programming,

* Everyone respects each other as a valued team member.
* Everyone contributes value such as enthusiasm.
* Developers respect the expertise of the customers and vice versa.
* Management respects the right of the developers to accept the responsibility and receive authority over their own work.

**Combined with communication, simplicity, and concrete feedback, courage becomes extremely valuable.**

* Communication supports courage because it opens the possibility for more high-risk, high-reward experiments.
* Simplicity supports courage because you can afford to be much more courageous with a simple system. You are much less likely to break it unknowingly.
* Courage supports simplicity because as soon as you see the possibility of simplifying the system you try it.
* Concrete feedback supports courage because you feel much safer trying radical modifications to the code, if you can see the tests turn green at the end. If any of the tests do not turn green, you know that you can throw the code away.

## Extreme Programming Principles

In Extreme Programming, the basic principles are derived from the values so that the development practices can be checked against these principles. Each principle embodies the values and is more concrete, i.e. rapid feedback − you either, have it or you do not.

**The fundamental principles of Extreme Programming are −**

* **Rapid feedback**
* **Assume simplicity**
* **Incremental change**
* **Embracing change**
* **Quality work**

### Rapid Feedback

Rapid feedback is to get the feedback, understand it, and put the learning back into the system as quickly as possible.

* The developers design, implement and test the system, and use that feedback in seconds or minutes instead of days, weeks, or months.
* The customers review the system to check how best it can contribute, and give feedback in days or weeks instead of months or years.

### Assume Simplicity

To assume simplicity is to treat every problem as if it can be solved with simplicity.

Traditionally, you are told to plan for the future, to design for reuse. The result of this approach may turn into ‘what is required today by the customer is not met and what is ultimately delivered may be obsolete and difficult to change.’

‘Assume Simplicity’ means ‘do a good job of solving today's job today and trust your ability to add complexity in the future where you need it.’ In Extreme Programming, you are told to do a good job (tests, refactoring, and communication) focusing on what is important today.

* With good unit tests, you can easily refactor your code to do additional tests.
* Follow YAGNI (You Ain’t Gonna Need It).
* Follow the DRY(Don’t Repeat Yourself) principle. For example,
  + Do not have multiple copies of identical (or very similar) code.
  + Do not have redundant copies of information.
  + No wastage of time and resources on what may not be necessary.

### Incremental Change

In any situation, big changes made all at once just do not work. Any problem is solved with a series of the smallest change that makes a difference.

In Extreme Programming, Incremental Change is applied in many ways.

* The design changes a little at a time.
* The plan changes a little at a time.
* The team changes a little at a time.

Even the adoption of Extreme Programming must be taken in little steps.

### Embracing Change

The best strategy is the one that preserves the most options while actually solving your most pressing problem.

### Quality Work

Everyone likes doing a good job. They try to produce the quality that they are proud of. The team

* **Works well**
* **Enjoys the work**
* **Feels good in producing a product of value**

There are four basic activities in Extreme Programming. They are −

* **Coding**
* **Testing**
* **Listening**
* **Designing**

These four basic activities need to be structured in the light of the Extreme Programming principles. To accomplish this, the Extreme Programming practices are defined.

## Four Areas of Extreme Programming

The Extreme Programming practices can be grouped into four areas −

* **Rapid, Fine Feedback −**
  + **Testing**
  + **On-site customer**
  + **Pair programming**
* **Continuous Process −**
  + **Continuous Integration**
  + **Refactoring**
  + **Short Releases**
* **Shared Understanding −**
  + **The Planning Game**
  + **Simple Design**
  + **Metaphor**
  + **Collective Ownership**
  + **Coding Standards**
* **Developer Welfare −**
  + **Forty-Hour Week**

### Extreme Programming Practices at-a-glance

The following diagram shows how Extreme Programming is woven around the Extreme Programming practices −



## The Planning Game

The main planning process within extreme programming is called the Planning Game. The game is a meeting that occurs once per iteration, typically once a week. The Planning Game is to Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.

Business and development need to make the decisions in tandem. The business decisions and the development’s technical decisions have to align with each other.

**Business people need to decide about −**

* **Scope** − How much of a problem must be solved for the system to be valuable in production? The businessperson is in a position to understand how much is not enough and how much is too much.
* **Priority** − If you are given an option, which one do you want? The businessperson is in a position to determine this, more than a developer with inputs from the customer.
* **Composition of releases** − How much or how little needs to be done before the business is better off with the software than without it? The developer's intuition about this question can be wildly wrong.
* **Dates of releases** − What are important dates at which the presence of the software (or some of the software) would make a big difference?

**Technical people need to decide about −**

* **Estimates** − How long will a feature take to implement?
* **Consequences** − There are strategic business decisions that should be made only when informed about the technical consequences. Development needs to explain the consequences.
* **Process** − How will the work and the team be organized? The team needs to fit the culture in which it will operate. The software must be written well rather than preserve the irrationality of an enclosing culture.
* **Detailed Scheduling** – With in a release, which stories should be done first? The developers need the freedom to schedule the riskiest segments of development first, to reduce the overall risk of the project. Within that constraint, they still tend to move business priorities earlier in the development, reducing the chance that important stories will have to be dropped towards the end of the development of a release due to time constraints.

Thus, plan is a result of collaboration between the customer, businessperson and the developers.

### The Planning Game – Advantages

**The Planning Game has the following advantages −**

* **Reduction in time, wasted on useless features**
* **Greater customer appreciation of the cost of a feature**
* **Less guesswork in planning**

## Short Releases

You should put a simple system into production quickly, and then release new versions in very short cycles. Every release should be as small as possible, so that it is −

* Achievable in a short cycle
* Contains the most valuable and immediate business requirements
* A working system

The duration of the short cycle may vary with the software that needs to be built. However, it needs to be ensured that the minimum possible duration is chosen.

### Short Releases – Advantages

**The advantages of Short Releases are −**

* Frequent feedback
* Tracking
* Reduce chance of overall project slippage

## Metaphor

A Metaphor is an expression, often found in literature that describes a person or object by referring to something that is considered to have similar characteristics to that person or object. For example, ‘The mind is an ocean’ and ‘the city is a jungle’ are both Metaphors.

You should guide the entire development with a simple shared story of how the whole system works.

The metaphor consists of domain specific elements and shows their interconnectivity. The language used is the domain language. To identify the technical entities, the words used in the metaphor need to be taken consistently.

As the development proceeds and the metaphor matures, the whole team will find new inspiration from examining the metaphor.

The goal of a good architecture is to give everyone a coherent story within which to work, a story that can easily be shared by both the business and the technical members. Hence, in Extreme Programming, by asking for a metaphor, we are likely to get an architecture that is easy to communicate and elaborate.

### Metaphor – Advantages

**The advantages of Metaphor are −**

* Encourages a common set of terms for the system
* Reduction of buzz words and jargon
* A quick and easy way to explain the system

## Simple Design

The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.

The right design for the software at any given time is the one that −

* Runs all the tests
* Has no duplicated logic like parallel class hierarchies
* States every intention important to the developers
* Has the fewest possible classes and methods

To get a simple design, eliminate any design element that you can, without violating the first three rules. This is opposite to the advice- Implement for today, design for tomorrow. If you believe that the future is uncertain and you can quickly enhance the design, then do not put any functionality on speculation.

### Simple Design – Advantages

**The advantages of Simple Design are −**

* Time is not wasted adding superfluous functionality
* Easier to understand what is going on
* Refactoring and collective ownership is made possible
* Helps keep the programmers on track

## Testing

The developers continually write unit tests, which need to pass for the development to continue. The customers write tests to verify that the features are implemented. The tests are automated so that they become a part of the system and can be continuously run to ensure the working of the system. The result is a system that is capable of accepting change.

### Testing – Advantages

**The advantages of testing are −**

* Unit testing promotes testing completeness
* Test-first gives developers a goal
* Automation gives a suite of regression tests

## Refactoring

When implementing a feature, the developers always ask if there is a way of changing the existing code to make adding the feature simple. After they have added a feature, the developers ask if they now can see how to make the code simpler, while still running all of the tests. They restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility. This is called Refactoring.

### Refactoring – Advantages

**The advantages of Refactoring are −**

* Prompts the developers to proactively improve the product as a whole
* Increases the developer knowledge of the system

## Pair Programming

In Pair programing, the entire code is written with two developers at one machine, with one keyboard and one mouse.

There are two roles in each pair −

* The first developer (the one with the keyboard and the mouse) thinks about the best way to implement this method right here.
* The other developer is thinks more strategically
  + Is this whole approach going to work?
  + What are some other test cases that might not work yet?
  + Is there some way to simplify the whole system so the current problem just disappears?

The pairing is dynamic. It means that the two Roles A and B may exchange their places, or they may pair up with other team members. More often, anyone on the team will do as a partner. For example, if you have a responsibility for a task in an area that is unfamiliar to you, you might ask someone with recent experience to pair with you.

### Pair Programming – Advantages

**The advantages of Pair Programming are −**

* Two heads are better than one
* Focus
* Two people are more likely to answer the following questions −
  + Is this whole approach going to work?
  + What are some test cases that may not work yet?
  + Is there a way to simplify this?

## Collective Ownership

In Extreme Programming, the entire team takes responsibility for the whole of the system. Not everyone knows every part equally well, although everyone knows something about every part.

If a pair is working and they see an opportunity to improve the code, they go ahead and improve it.

### Collective Ownership – Advantages

**The advantages of Collective ownership are −**

* Helps mitigate the loss of a team member who is leaving.
* Promotes the developers to take responsibility for the system as a whole rather than parts of the system.

## Continuous Integration

Code is integrated and tested many times a day, one set of changes at a time. A simple way to do this is to have a machine dedicated to integration. A pair with code ready to integrate −

* Sits when the machine is free.
* Loads the current release.
* Loads their changes (checking for and resolving any collisions).
* Runs the tests until they pass (100% correct).

Integrating one set of changes at a time helps in knowing who should fix a test that fails. The answer- is the present pair, since the last pair left the tests at 100%. They may have to throw away what they have done and start all over, as they might not have known enough to code that feature.

### Continuous Integration – Advantages

**The advantages of Continuous Integration are −**

* Reduces the duration, which is otherwise lengthy.
* Enables the short releases practice as the time required before release is minimal.

## 40-Hour Week

Extreme Programming emphasizes on the limited number of hours of work per week for every team members, based on their sustainability, to a maximum of 45 hours a week. If someone works for more time than that, it is considered as overtime. Overtime is allowed for at most one week. This practice is to ensure that every team member be fresh, creative, careful and confident.

### 40-Hour Week – Advantages

**The advantages of 40-hour week are −**

* Most developers lose effectiveness past 40 hours.
* Value is placed on the developers’ well-being.
* Management is forced to find real solutions.

## On-Site Customer

Include a real, live user on the team, available full-time to answer the questions, resolve disputes and set small-scale priorities. This user may not have to spend 40 hours on this role only and can focus on other work too.

### On-Site Customer – Advantages

The advantages of having an onsite customer are −

* Can give quick and knowledgeable answers to the real development questions.
* Makes sure that what is developed is what is needed.
* Functionality is prioritized correctly.

## Coding Standards

**Developers write all code in accordance with the rules emphasizing-**

* Communication through the code.
* The least amount of work possible.
* Consistent with the “once and only once” rule (no duplicate code).
* Voluntary adoption by the whole team.

**These rules are necessary in Extreme Programming because all the developers −**

* Can change from one part of the system to another part of the system.
* Swap partners a couple of times a day.
* Refactor each other's code constantly.

If the rules are not followed, the developers will tend to have different sets of coding practices, the code becomes inconsistent over time and it becomes impossible to say who on the team wrote what code.

### Coding Standards – Advantages

**The advantages of Coding Standards are −**

* Reduces the amount of time developers spend reformatting other peoples’ code.
* Reduces the need for internal commenting.
* Calls for clear, unambiguous code.

# **Extreme Programming: Values, Principles, and Practices**

With software engineering existing in such a fast-paced environment, traditional project management approaches are no longer viable. That means that IT professionals must find new ways to handle frequently changing development tasks.

**Extreme Programming (XP)** is one of the numerous Agile frameworks applied by IT companies. But its key feature — emphasis on technical aspects of software development — distinguishes XP from the other approaches.

**Software engineer**[**Ken Beck**](https://en.wikipedia.org/wiki/Kent_Beck)**introduced XP in the 90s with the goal of finding ways to write high-qualitative software quickly and being able to adapt to customers’ changing requirements. In 1999, he refined XP approaches in the book**[***Extreme Programming Explained: Embrace Change***](https://www.amazon.com/Extreme-Programming-Explained-Embrace-Change/dp/0201616416)**.**

## The process and roles of extreme programming

**The XP framework normally involves 5 phases or stages of the development process that**[**iterate**](https://www.altexsoft.com/blog/iterative-process-agile/)**continuously:**

1. **Planning,**the first stage, is when the customer meets the development team and presents the [requirements](https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/) in the form of [user stories](https://www.altexsoft.com/blog/user-stories/) to describe the desired result. The team then [estimates](https://www.altexsoft.com/blog/story-points/) the stories and creates a release plan broken down into iterations needed to cover the required functionality part after part. If one or more of the stories can’t be estimated, so-called spikes can be introduced which means that further research is needed.
2. **Designing**is actually a part of the planning process, but can be set apart to emphasize its importance. It’s related to one of the main XP values that we’ll discuss below — simplicity. A good design brings logic and structure to the system and allows to avoid unnecessary complexities and redundancies.
3. **Coding**is the phase during which the actual code is created by implementing specific XP practices such as coding standards, pair programming, continuous integration, and collective code ownership (the entire list is described below).
4. **Testing**is the core of extreme programming. It is the regular activity that involves both unit tests ([automated testing](https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/) to determine if the developed feature works properly) and acceptance tests (customer testing to verify that the overall system is created according to the initial requirements).
5. **Listening**is all about constant communication and feedback. The customers and project managers are involved to describe the business logic and value that is expected.
6. Logic and value that is expected.

*XP lifecycle* Such a development process entails the cooperation between several participants, each having his or her own tasks and responsibilities. Extreme programming puts people in the center of the system, emphasizing the value and importance of such social skills as communication, cooperation, responsiveness, and feedback . So, these roles are commonly associated with XP:

1. **Customers**are expected to be heavily engaged in the development process by creating user stories, providing continuous feedback, and making all the necessary business decisions related to the project.
2. **Programmers or developers**are the team members that actually create the product. They are responsible for implementing user stories and conducting user tests (sometimes a separate **Tester** role is set apart). Since XP is usually associated with [cross-functional teams](https://www.altexsoft.com/blog/cross-functional-teams/), the skill set of such members can be different.
3. **Trackers or managers** link customers and developers. It’s not a required role and can be performed by one of the developers. These people organize the meetups, regulate discussions, and keep track of important progress KPIs.
4. **Coaches** can be included in the teams as mentors to help with understanding the XP practices. It’s usually an outside assistant or external consultant who is not involved in the development process, but has used XP before and so can help avoid mistakes.

## Values and principles of extreme programming

In the late 90s, Ken Beck summarized a set of certain values and principles that describe extreme programming and lead to more effective cooperation within the team and, ultimately, higher product quality.

### Values of extreme programming

XP has simple rules that are based on 5 values to guide the teamwork:

1. **Communication.** Everyone on a team works jointly at every stage of the project.
2. **Simplicity.** Developers strive to write simple code bringing more value to a product, as it saves time and effort.
3. **Feedback.** Team members deliver software frequently, get feedback about it, and improve a product according to the new requirements.
4. **Respect.** Every person assigned to a project contributes to a common goal.
5. **Courage.** Programmers objectively evaluate their own results without making excuses and are always ready to respond to changes.

These values represent a specific mind set of motivated team players who do their best on the way to achieving a common goal. XP principles derive from these values and reflect them in more concrete ways.

### Principles of extreme programming

Most researchers denote 5 XP principles as:

1. **Rapid feedback.** Team members understand the given feedback and react to it right away.
2. **Assumed simplicity.** Developers need to focus on the job that is important at the moment and follow YAGNI (You Ain’t Gonna Need It) and DRY (Don’t Repeat Yourself) principles.
3. **Incremental changes.** Small changes made to a product step by step work better than big ones made at once.
4. **Embracing change.** If a client thinks a product needs to be changed, programmers should support this decision and plan how to implement new requirements.
5. **Quality work.** A team that works well, makes a valuable product and feels proud of it.

Having discussed the main values and principles of XP, let’s take a closer look at the practices inherent in this framework.

## Extreme programming practices

The practices of XP are a set of specific rules and methods that distinguishes it from other methodologies. When used in conjunction, they reinforce each other, help mitigate the risks of the development process, and lead to the expected high-quality result. XP suggests using 12 practices while developing software which can be clustered into four groups.

### Test-Driven Development

Is it possible to write a clear code quickly? The answer is yes, according to XP practitioners. The quality of software derives from short development cycles that, in turn, allow for receiving frequent feedback. And valuable feedback comes from good testing. XP teams practice test-driven development technique (TDD) that entails writing an automated unit test before the code itself. According to this approach, every piece of code must pass the test to be released. So, software engineers thereby focus on writing code that can accomplish the needed function. That’s the way TDD allows programmers to use immediate feedback to produce reliable software. You can learn more about [improving software testing](https://www.altexsoft.com/blog/engineering/software-testing-qa-best-practices/) in our dedicated article.

### The Planning Game

This is a meeting that occurs at the beginning of an [iteration cycle](https://www.altexsoft.com/blog/iterative-process-agile/). The development team and the customer get together to discuss and approve a product’s features. At the end of the planning game, developers plan for the upcoming iteration and release, assigning tasks for each of them.

### On-site Customer

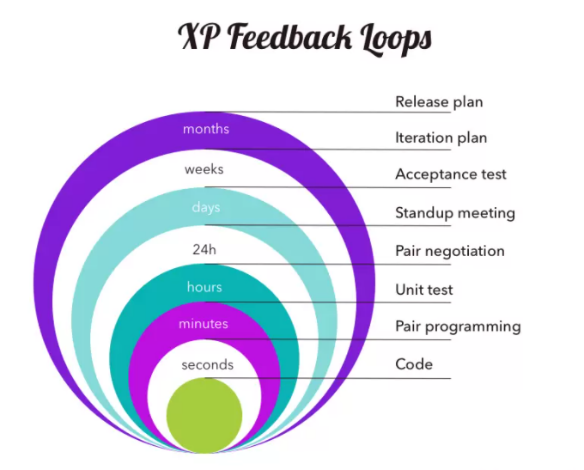
As we already mentioned, according to XP, the end customer should fully participate in development. The customer should be present all the time to answer team questions, set priorities, and resolve disputes if necessary.

### Pair Programming

This practice requires two programmers to work jointly on the same code. While the first developer focuses on writing, the other one reviews code, suggests improvements, and fixes mistakes along the way. Such teamwork results in high-quality software and faster knowledge sharing but takes about [15 percent more time](https://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf). In this regard, it’s more reasonable trying pair programming for long-term projects.

### Code Refactoring

To deliver business value with well-designed software in every short iteration, XP teams also use refactoring. The goal of this technique is to continuously improve code. Refactoring is about removing redundancy, eliminating unnecessary functions, increasing code coherency, and at the same time decoupling elements. Keep your code clean and simple, so you can easily understand and modify it when required would be the advice of any XP team member.



Pair programming in XP iteration cycle

### Continuous Integration

Developers always keep the system fully integrated. XP teams take iterative development to another level because they commit code multiple times a day, which is also called [continuous delivery](https://www.altexsoft.com/blog/business/continuous-delivery-and-integration-rapid-updates-by-automating-quality-assurance/). XP practitioners understand the importance of communication. Programmers discuss which parts of the code can be re-used or shared. This way, they know exactly what functionality they need to develop. The policy of shared code helps eliminate integration problems. In addition, automated testing allows developers to detect and fix errors before deployment.

### Small Releases

This practice suggests releasing the [MVP](https://www.altexsoft.com/blog/business/minimum-viable-product-types-methods-and-building-stages/) quickly and further developing the product by making small and incremental updates. Small releases allow developers to frequently receive feedback, detect bugs early, and monitor how the product works in production. One of the methods of doing so is the continuous integration practice (CI) we mentioned before.

### Simple Design

The best design for software is the simplest one that works. If any complexity is found, it should be removed. The right design should pass all tests, have no duplicate code, and contain the fewest possible methods and classes. It should also clearly reflect the programmer’s intent.

XP practitioners highlight that chances to simplify design are higher after the product has been in production for some time. [Don Wells](http://www.extremeprogramming.org/rules/simple.html) advises writing code for those features you plan to implement right away rather than writing it in advance for other future features: “The best approach is to create code only for the features you are implementing while you search for enough knowledge to reveal the simplest design. Then refactor incrementally to implement your new understanding and design.”

### Coding Standards

A team must have common sets of coding practices, using the same formats and styles for code writing. Application of standards allows all team members to read, share, and refactor code with ease, track who worked on certain pieces of code, as well as make the learning faster for other programmers. Code written according to the same rules encourages collective ownership.

### Collective Code Ownership

This practice declares a whole team’s responsibility for the design of a system. Each team member can review and update code. Developers that have access to code won’t get into a situation in which they don’t know the right place to add a new feature. The practice helps avoid code duplication. The implementation of collective code ownership encourages the team to cooperate more and feel free to bring new ideas.

### System Metaphor

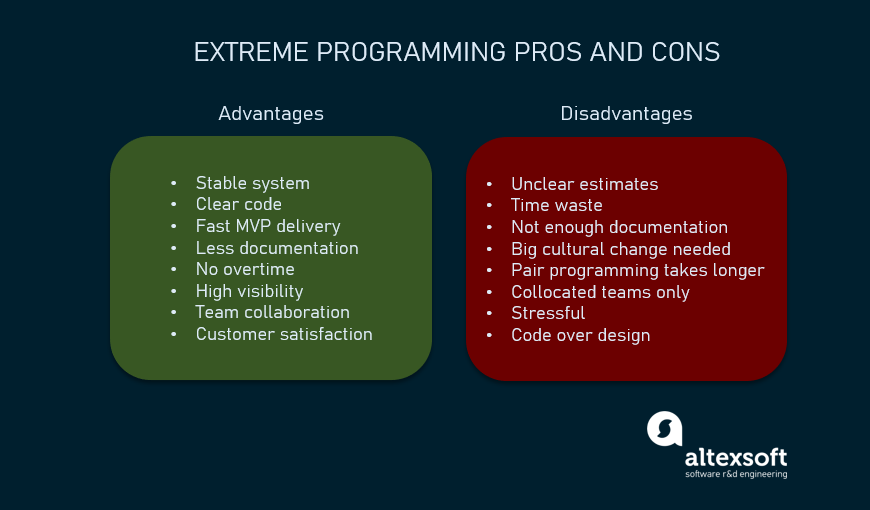
System metaphor stands for a simple design that has a set of certain qualities. First, a design and its structure must be understandable to new people. They should be able to start working on it without spending too much time examining specifications. Second, the naming of classes and methods should be coherent. Developers should aim at naming an object as if it already existed, which makes the overall system design understandable.

### 40-Hour Week

XP projects require developers to work fast, be efficient, and sustain the product’s quality. To adhere to these requirements, they should feel well and rested. Keeping the work-life balance prevents professionals from burnout. In XP, the optimal number of work hours must not exceed 45 hours a week. One overtime a week is possible only if there will be none the week after.

## Advantages and disadvantages of XP

XP practices have been debated upon for decades, as its approach and methods are rather controversial in a number of aspects and can’t be applied in just any project. Here, we’ll try to define the pros and cons of XP methodology.



XP pros and cons

### Extreme programming advantages

So, the XP framework can be beneficial and help reduce development time and costs for the following reasons:

* Continuous testing and refactoring practices help create **stable well-performing systems** with minimal debugging;
* Simplicity value implies creating a **clear, concise code** that is easy to read and change in the future if needed;
* The minimalistic iterative approach to development ensures that the workable **results can be delivered very soon** and only necessary features are built;
* **Documentation is reduced** as bulky [requirements documents](https://www.altexsoft.com/blog/business/technical-documentation-in-software-development-types-best-practices-and-tools/) are substituted by user stories;
* No or **very little overtime** is practiced;
* Constant communication provides a high level of **visibility and accountability** and allows all team members to keep up with the project progress;
* Pair programming has [proven](https://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf) to result in **higher-quality products** with fewer bugs; most research participants also reported enjoying such collaboration more and feeling more confident about their job;
* **Customer engagement ensures their satisfaction** as their participation in the development and testing process can directly influence the result, getting them exactly what they wanted.

### Extreme programming disadvantages

On the other hand, XP has a number of disadvantages that have to be considered when deciding on which framework to choose for your next project:

* In many instances, the customer has no clear picture of the end result, which makes it almost **unrealistic to accurately estimate scope, cost, and time**;
* **Regular meetings with customers often take a great deal of time** that could instead be spent on actual code writing;
* **Documentation can be scarce** and lack clear requirements and specifications, leading to project scope creep;
* The rapid transition from traditional methods of software development to extreme programming demands significant **cultural and structural changes**;
* **Pair programming takes more time** and doesn’t always work right due to the human factor and character incompatibility;
* **XP works best with collocated teams** and customers present in person to conduct face-to-face meetings, limiting its application with distributed teams;
* Sometimes customers have neither the desire, time, nor expertise to participate in product development. Considering tight deadlines, it can become a **source of stress** as either no valuable feedback is provided, or a non-technical representative attempts to manage tech specialists with little or no knowledge on the process;
* Some authors also mention overfocusing on code over design, lack of quality assurance, code duplication, and poor results with inexperienced developers.

Any company can apply the XP principles in its projects; however, it’s important to understand both the good and the bad sides. Read on to find out how XP is different from other methodologies and when applying its techniques would be the best choice.

## When to use XP

Now that we discussed the XP methodology pros and cons and identified its place among other agile frameworks, we can talk about the cases when it’s applicable. It’s important to make sure a company’s size, structure, and expertise, as well as the staff’s knowledge base allow for applying XP practices. These are the factors to consider.

**Highly-adaptive development.** Some systems don’t have constant functionality features and implies frequent changes. XP was designed to help development teams adapt to fast-changing requirements.

**Risky projects.** Teams applying XP practices are more likely to avoid problems connected with working on a new system, especially when a customer sets strict deadlines for a project. Additionally, a high level of customer engagement reduces the risk of their not accepting the end product.

**Small teams.** XP practices are efficient for teams that don’t exceed 12 people. Managing such groups is usually easier, communication is more efficient, and it takes less time to conduct meetings and brainstorming sessions.

**Automated testing.** Another factor that can influence the choice of XP is the developers’ ability to create and run unit tests, as well as availability of the necessary testing tools.

**Readiness to accept new culture and knowledge.** XP is different from traditional approaches to software development, and the way some of its practices should be implemented might not be obvious. So, it’s important that your organization and team members are ready to embrace change. It’s also worth inviting an experienced coach if you don’t have previous involvement with XP.

**Customer participation.** As XP requires customers, developers and managers to work side-by-side, make sure your client is always available to provide input until a project ends.

Agility principles are becoming [increasingly popular](https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494) as they prove their effectiveness. Even though extreme programming is not the most widespread methodology, it offers a lot of sensible practices that can benefit software development and are worth considering for implementation in your projects.

**Exercise 3:**

**It is important to comprehend the need to automate the software development lifecycle stages through DevOps. Gain an understanding of the capabilities required to implement DevOps, continuous integration and continuous delivery practices.**

## What Is DevOps Lifecycle?

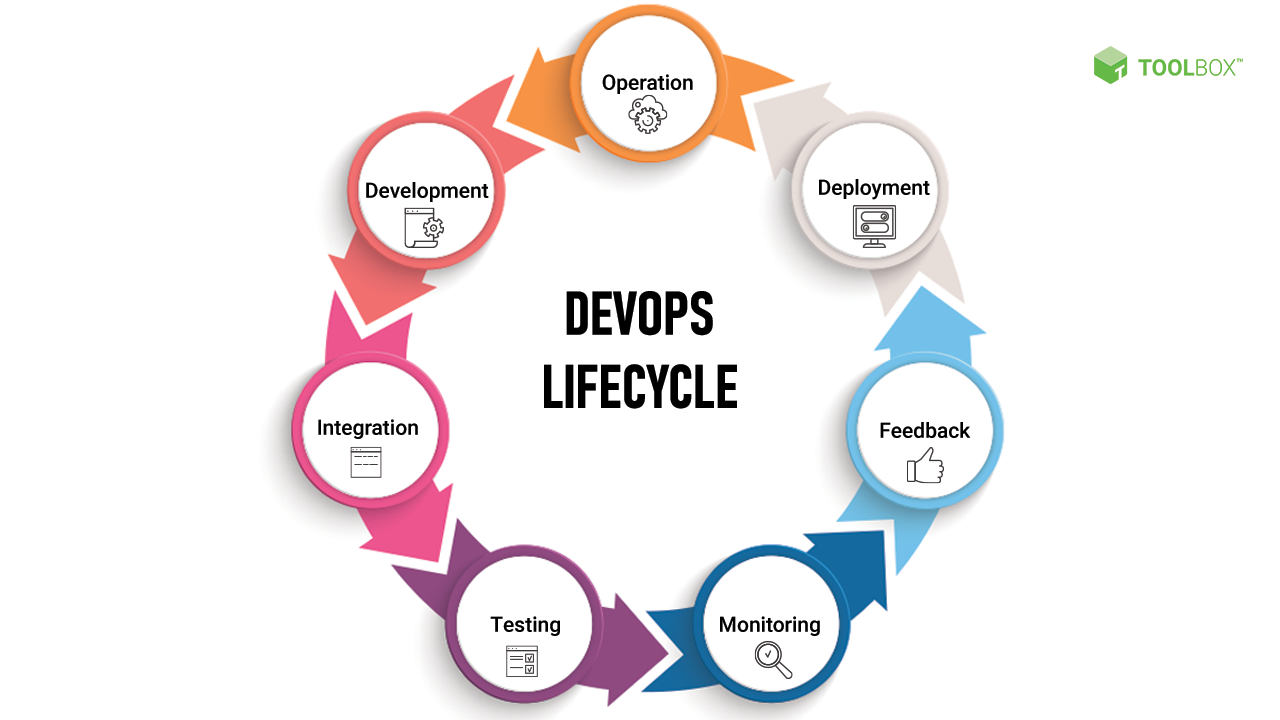
[**DevOps**](https://www.spiceworks.com/tech/devops/articles/what-is-devops/)**lifecycle is a combination of different phases of continuous software development, integration, testing, deployment, and monitoring. A competent DevOps lifecycle is necessary to leverage the full benefits of the DevOps methodology.**

The DevOps approach embraces continuous innovation, agility, and scalability to build, test, consume, and evolve software products. It promotes a culture of experimentation, feedback, and constant learning to reinvent products, services, and processes. However, to implement DevOps, a proper understanding of different phases of the DevOps lifecycle is crucial.

To deliver faster results, developers must be fully aware of all the different phases of the DevOps lifecycle. If they aren’t, the entire development process can become complex and time-consuming. Here is a complete breakdown and analysis of each component of the DevOps lifecycle.

**DevOps Lifecycle: Key Components**

The DevOps lifecycle optimizes development processes from start to end and engages the organization in continuous development, resulting in faster delivery times. This process mainly consists of the following seven stages.



**DevOps Lifecycle**

**1. Continuous development**

Continuous development involves planning and [coding](https://www.spiceworks.com/tech/devops/articles/developer-tips-for-secure-coding/) the software. Here, the entire development process gets broken down into smaller development cycles. This process makes it easier for the DevOps team to accelerate the overall software development process. This phase is instrumental in mapping the vision for the entire development cycle, enabling developers to fully understand project expectations. Through this, the team starts visualizing its end goal as well.

There are no DevOps tools required for [planning](https://www.spiceworks.com/tech/devops/articles/devops-strategies-to-test/), but many version control tools are used to maintain code. This process of code maintenance is called source code maintenance. Popular tools for source code maintenance include JIRA, Git, Mercurial, and SVN. Moreover, there are different tools for packaging the codes into executable files, such as Ant, Gradle, and Maven. These executable files are then forwarded to the next component of the DevOps lifecycle.

**2. Continuous integration**

Continuous integration (CI) includes different steps related to the execution of the test process. Along with this, clients also provide information to be incorporated for adding new features to the application. Most changes happen in the source code during this phase. CI becomes the hub for resolving these frequent changes on a daily or monthly basis. Building code is a combination of unit and integration testing, code review, and packaging. Since [developers](https://www.spiceworks.com/tech/devops/articles/developer-tips-for-secure-coding/) make frequent changes, they can quickly spot problems (if any) and resolve them at an early stage.

This phase experiences continuous integrations of new code functionalities with the existing source code. Due to continuous development, the updated code seamlessly integrates within the entire system. Jenkins is one of the most popular tools for continuous integration. It helps in fetching the updated code and preparing an executable build.

**3. Continuous testing**

The DevOps lifecycle is the testing phase, wherein the developed code is tested for bugs and errors that may have made their way into the code. This is where quality analysis (QA) plays a major role in checking the usability of the developed software. Successful completion of the QA process is crucial in determining whether the software meets the client’s specifications.

Automation tools, such as JUnit, Selenium, and TestNG, are used for continuous testing, enabling the QA team to analyze multiple code-bases simultaneously. Doing this ensures that there are no flaws in the functionality of the developed software.

Moreover, to simulate the entire test environment, Docker containers are used in [continuous testing](https://www.spiceworks.com/tech/devops/guest-article/the-devops-tightrope-balancing-shifting-left-and-right/). A Docker container is a standalone, lightweight executable package with everything to run an app: system tools, system libraries, runtime code, and settings.

Automated testing is done on automation tools like Selenium, after which the reports are generated on another automation tool, for example, TestNG. Automation of the entire testing phase also becomes possible with the help of the continuous integration tool Jenkins. Automation testing plays a vital role in saving time, labor and effort.

**4. Continuous deployment**

Continuous deployment (CD) ensures hassle-free product deployment without affecting the application’s performance. It is necessary to ensure that the code is deployed precisely on all available servers during this phase. This process eliminates the need for scheduled releases and accelerates the feedback mechanism, allowing developers to address issues more quickly and with greater accuracy.

Containerization tools help achieve continuous deployment through configuration management. A containerization tool like Vagrant helps achieve consistency across test, development, staging, and production environments. [Containerization](https://www.spiceworks.com/security/cloud-security/articles/what-is-container-security-definition-components/) deals with bringing virtualization to the level of an operating system.

Continuous deployment is guaranteed to benefit your organization once you have a reliable automated testing environment in place. Configuration management holds a lot of value in the continuous deployment phase. It involves configuring and maintaining consistency in the functional requirement of the app. Popular DevOps tools used for configuration management include Ansible, Puppet, and Chef that help execute quick deployment of new code.

**5. Continuous monitoring**

[Monitoring](https://www.spiceworks.com/it-security/application-security/articles/is-application-performance-monitoring-key-to-protecting-critical-infrastructure-against-cyberattacks/) the performance of a software product is essential to determine the overall efficacy of the product output. This phase processes important information about the developed app. Through continuous monitoring, developers can identify general patterns and gray areas in the app where more effort is required.

Continuous monitoring is an operational phase where the objective is to enhance the overall efficiency of the software application. Moreover, it monitors the performance of the app as well. Therefore, it is one of the most crucial phases of the DevOps lifecycle.

Different [system errors](https://www.spiceworks.com/tech/devops/articles/chaos-engineering-test-system-resilience-before-a-disaster-strikes/) such as ‘server not reachable’, ‘low memory’, etc., are resolved in the continuous monitoring phase. It also maintains the availability and security of the services. Network issues and other problems are automatically fixed during this phase at the time of their detection.

Tools such as Nagios, Splunk, Sensu, ELK Stack, and NewRelic are used by the operations team to monitor user activities for improper behavior. As a result, during continuous monitoring, developers can proactively check the overall health of the system.

Proactive checking improves the [reliability](https://www.spiceworks.com/it-security/data-security/guest-article/whos-responsible-for-data-protection-in-the-wfh-era-it-or-devops/) and productivity of the system and also reduces maintenance costs. Moreover, important and major issues are directly reported to the development team to be corrected in the initial stages. This leads to faster resolution of issues**.**

**6. Continuous feedback**

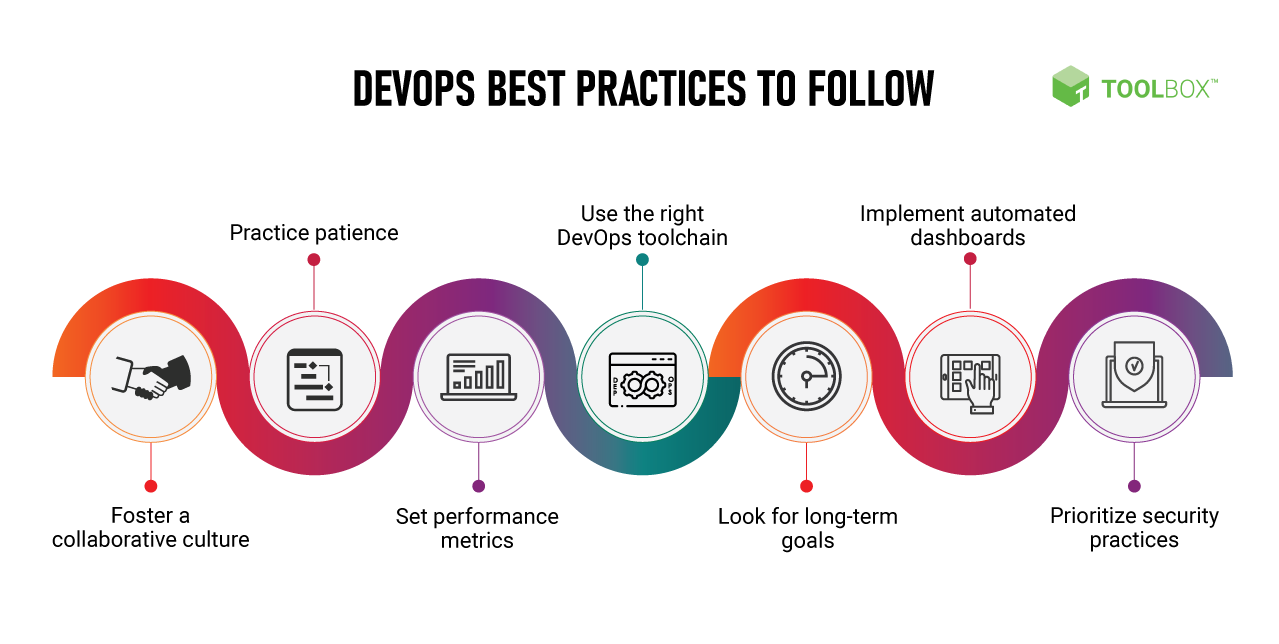
Continuous feedback is essential to ascertain and analyze the final outcome of the application. It sets the tone for improving the current version and releasing a new version based on stakeholder feedback.

The overall process of app development can only be improved by analyzing the results from the software operations. Feedback is nothing but information gathered from the client’s end. Here, information is significant, as it carries all the data about the performance of the software and its related issues. It also contains suggestions given by end users of the software.

**7. Continuous operations**

The last stage in the DevOps lifecycle is the shortest and easiest to grasp. Continuity is at the heart of all DevOps operations that helps automate release processes, allows developers to detect issues quickly, and build better versions of software products. Continuation is key to eliminate diversions and other extra steps that hinder development.

Development cycles in continuous operations are shorter, allowing organizations to advertise constantly and accelerate the overall time to market the product. [DevOps](https://www.spiceworks.com/tech/devops/articles/why-agile-devops-is-now-the-default-standard-for-software-development/" \o "DevOps) enhances the value of software products by making them better and more efficient, thereby attracting new customers towards it.



**DevOps capabilities**

The [DevOps Research and Assessment (DORA)](https://dora.dev/) team has identified and validated a set of capabilities that drive higher software delivery and organizational performance. These articles describe how to implement, improve, and measure these capabilities.

**Technical capabilities**

**[Cloud infrastructure](https://dora.dev/devops-capabilities/technical/cloud-infrastructure)**

[Find out how to manage cloud infrastructure effectively so you can achieve higher levels of agility, availability, and cost visibility.](https://dora.dev/devops-capabilities/technical/cloud-infrastructure)

**[Code maintainability](https://dora.dev/devops-capabilities/technical/code-maintainability)**

[Make it easy for developers to find, reuse, and change code, and keep dependencies up-to-date.](https://dora.dev/devops-capabilities/technical/code-maintainability)

**[Continuous delivery](https://dora.dev/devops-capabilities/technical/continuous-delivery)**

[Make deploying software a reliable, low-risk process that can be performed on demand at any time.](https://dora.dev/devops-capabilities/technical/continuous-delivery)

**[Continuous integration](https://dora.dev/devops-capabilities/technical/continuous-integration)**

[Learn about common mistakes, ways to measure, and how to improve on your continuous integration efforts.](https://dora.dev/devops-capabilities/technical/continuous-integration)

**Continuous testing**

Improve software quality by building reliable automated test suites and performing all kinds of testing throughout the software delivery lifecycle.

**[Database change management](https://dora.dev/devops-capabilities/technical/database-change-management)**

[Make sure database changes don't cause problems or slow you down.](https://dora.dev/devops-capabilities/technical/database-change-management)

**[Deployment automation](https://dora.dev/devops-capabilities/technical/deployment-automation)**

[Best practices and approaches for deployment automation and reducing manual intervention in the release process.](https://dora.dev/devops-capabilities/technical/deployment-automation)

**[Empowering teams to choose tools](https://dora.dev/devops-capabilities/technical/teams-empowered-to-choose-tools)**

[Empower teams to make informed decisions on tools and technologies. Learn how these decisions drive more effective software delivery.](https://dora.dev/devops-capabilities/technical/teams-empowered-to-choose-tools)

**[Loosely coupled architecture](https://dora.dev/devops-capabilities/technical/loosely-coupled-architecture/)**

[Learn about moving from a tightly coupled architecture to service-oriented and microservice architectures without re-architecting everything at once.](https://dora.dev/devops-capabilities/technical/loosely-coupled-architecture/)

**[Monitoring and observability](https://dora.dev/devops-capabilities/technical/monitoring-and-observability)**

[Learn how to build tooling to help you understand and debug your production systems.](https://dora.dev/devops-capabilities/technical/monitoring-and-observability)

**[Shifting left on security](https://dora.dev/devops-capabilities/technical/shifting-left-on-security)**

[Build security into the software development lifecycle without compromising delivery speed.](https://dora.dev/devops-capabilities/technical/shifting-left-on-security)

**[Test data management](https://dora.dev/devops-capabilities/technical/test-data-management)**

[Understand the right strategies for managing test data effectively along with approaches to provide fast, secure data access for testing.](https://dora.dev/devops-capabilities/technical/test-data-management)

**[Trunk-based development](https://dora.dev/devops-capabilities/technical/trunk-based-development)**

[Prevent merge-conflict hassles with trunk-based development practices.](https://dora.dev/devops-capabilities/technical/trunk-based-development)

**[Version control](https://dora.dev/devops-capabilities/technical/version-control)**

[A guide to implementing the right version control practices for reproducibility and traceability.](https://dora.dev/devops-capabilities/technical/version-control)

**Process capabilities**

**[Customer feedback](https://dora.dev/devops-capabilities/process/customer-feedback)**

[Drive better organizational outcomes by gathering customer feedback and incorporating it into product and feature design.](https://dora.dev/devops-capabilities/process/customer-feedback)

**[Monitoring systems to inform business decisions](https://dora.dev/devops-capabilities/process/monitoring-systems)**

[Improve monitoring across infrastructure platforms, middleware, and the application tier, so you can provide fast feedback to developers.](https://dora.dev/devops-capabilities/process/monitoring-systems)

**[Proactive failure notification](https://dora.dev/devops-capabilities/process/proactive-failure-notification)**

[Set proactive failure notifications to identify critical issues and act on problems before they arise.](https://dora.dev/devops-capabilities/process/proactive-failure-notification)

**[Streamlining change approval](https://dora.dev/devops-capabilities/process/streamlining-change-approval)**

[Replace heavyweight change-approval processes with peer review, to get the benefits of a more reliable, compliant release process without sacrificing speed.](https://dora.dev/devops-capabilities/process/streamlining-change-approval)

**[Team experimentation](https://dora.dev/devops-capabilities/process/team-experimentation)**

[Innovate faster by building empowered teams that can try out new ideas without approval from people outside the team.](https://dora.dev/devops-capabilities/process/team-experimentation)

**[Visibility of work in the value stream](https://dora.dev/devops-capabilities/process/work-visibility-in-value-stream)**

[Understand and visualize the flow of work from idea to customer outcome in order to drive higher performance.](https://dora.dev/devops-capabilities/process/work-visibility-in-value-stream)

**[Visual management capabilities](https://dora.dev/devops-capabilities/process/visual-management%7B:.external%7D)**

[Learn about the principles of visual management to promote information sharing, get a common understanding of where the team is, and how to improve.](https://dora.dev/devops-capabilities/process/visual-management%7B:.external%7D)

**[Work in process limits](https://dora.dev/devops-capabilities/process/wip-limits)**

[Prioritize work, limit the amount of things that people are working on, and focus on getting a small number of high-priority tasks done.](https://dora.dev/devops-capabilities/process/wip-limits)

**[Working in small batches](https://dora.dev/devops-capabilities/process/working-in-small-batches)**

[Create shorter lead times and faster feedback loops by working in small batches. Learn common obstacles to this critical capability and how to overcome them.](https://dora.dev/devops-capabilities/process/working-in-small-batches)

**Cultural capabilities**

**[Generative organizational culture](https://dora.dev/devops-capabilities/cultural/generative-organizational-culture)**

[Discover how growing a generative, high-trust culture drives better organizational and software delivery performance.](https://dora.dev/devops-capabilities/cultural/generative-organizational-culture)

**[Job satisfaction](https://dora.dev/devops-capabilities/cultural/job-satisfaction)**

[Find out about the importance of ensuring your people have the tools and resources to do their job, and of making good use of their skills and abilities.](https://dora.dev/devops-capabilities/cultural/job-satisfaction)

**[Learning culture](https://dora.dev/devops-capabilities/cultural/learning-culture)**

[Grow a learning culture and understand its impact on your organizational performance.](https://dora.dev/devops-capabilities/cultural/learning-culture)

**[Transformational leadership](https://dora.dev/devops-capabilities/cultural/transformational-leadership)**

[Learn how effective leaders influence software delivery performance by driving the adoption of technical and product management capabilities.](https://dora.dev/devops-capabilities/cultural/transformational-leadership)

Continuous integration

Continuous integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. CI most often refers to the build or integration stage of the software release process and requires both an automation component (for example a CI or build service) and a cultural component (for example learning to integrate frequently). The key goals of CI are to find and address bugs more quickly, improve software quality, and reduce the time it takes to validate and release new software updates.

Continuous integration focuses on smaller commits and smaller code changes to integrate. A developer commits code at regular intervals, at minimum once a day. The developer pulls code from the code repository to ensure the code on the local host is merged before pushing to the build server. At this stage the build server runs the various tests and either accepts or rejects the code commit.

The basic challenges of implementing CI include more frequent commits to the common codebase, maintaining a single source code repository, automating builds, and automating testing. Additional challenges include testing in similar environments to production, providing visibility of the process to the team, and allowing developers to easily obtain any version of the application.

Continuous delivery and deployment

Continuous delivery (CD) is a software development practice where code changes are automatically built, tested, and prepared for production release. It expands on continuous integration by deploying all code changes to a testing environment, a production environment, or both after the build stage has been completed. Continuous delivery can be fully automated with a workflow process or partially automated with manual steps at critical points. When continuous delivery is properly implemented, developers always have a deployment-ready build artifact that has passed through a standardized test process.

With continuous deployment, revisions are deployed to a production environment automatically without explicit approval from a developer, making the entire software release process automated. This, in turn, allows for a continuous customer feedback loop early in the product lifecycle.

Continuous delivery is not continuous deployment

One misconception about continuous delivery is that it means every change committed is applied to production immediately after passing automated tests. However, the point of continuous delivery is not to apply every change to production immediately, but to ensure that every change is ready to go to production.

Before deploying a change to production, you can implement a decision process to ensure that the production deployment is authorized and audited. This decision can be made by a person and then executed by the tooling.

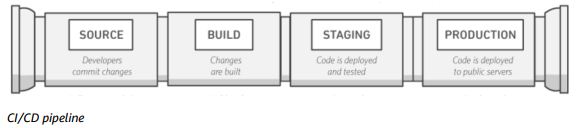
Using continuous delivery, the decision to go live becomes a business decision, not a technical one. The technical validation happens on every commit.

Rolling out a change to production is not a disruptive event. Deployment doesn’t require the technical team to stop working on the next set of changes, and it doesn’t need a project plan, handover documentation, or a maintenance window. Deployment becomes a repeatable process that has been carried out and proven multiple times in testing environments.

**Practicing Continuous Integration and Continuous Delivery**

**Implementing continuous integration and continuous delivery**

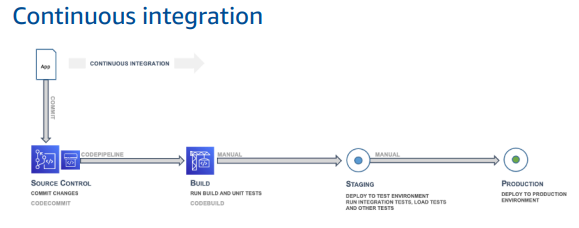
A pathway to continuous integration/continuous delivery CI/CD can be pictured as a pipeline (refer to the following figure), where new code is submitted on one end, tested over a series of stages (source, build, staging, and production), and then published as production-ready code. If your organization is new to CI/CD it can approach this pipeline in an iterative fashion. This means that you should start small, and iterate at each stage so that you can understand and develop your code in a way that will help your organization grow.

****Each stage of the CI/CD pipeline is structured as a logical unit in the delivery process. In addition, each stage acts as a gate that vets a certain aspect of the code. As the code progresses through the pipeline, the assumption is that the quality of the code is higher in the later stages because more aspects of it continue to be verified. Problems uncovered in an early stage stop the code from progressing through the pipeline. Results from the tests are immediately sent to the team, and all further builds and releases are stopped if software does not pass the stage.

These stages are suggestions. You can adapt the stages based on your business need. Some stages can be repeated for multiple types of testing, security, and performance. Depending on the complexity of your project and the structure of your teams, some stages can be repeated several times at different levels. For example, the end product of one team can become a dependency in the project of the next team. This means that the first team’s end product is subsequently staged as an artifact in the next team’s project.

The presence of a CI/CD pipeline will have a large impact on maturing the capabilities of your organization. The organization should start with small steps and not try to build a fully mature pipeline, with multiple environments, many testing phases, and automation in all stages at the start. Keep in mind that even organizations that have highly mature CI/CD environments still need to continuously improve their pipelines.

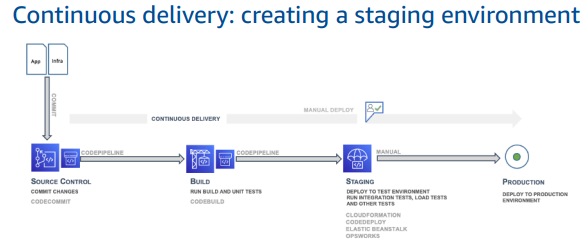
Building a CI/CD-enabled organization is a journey, and there are many destinations along the way. The next section discusses a possible pathway that your organization could take, starting with continuous integration through the levels of continuous delivery.

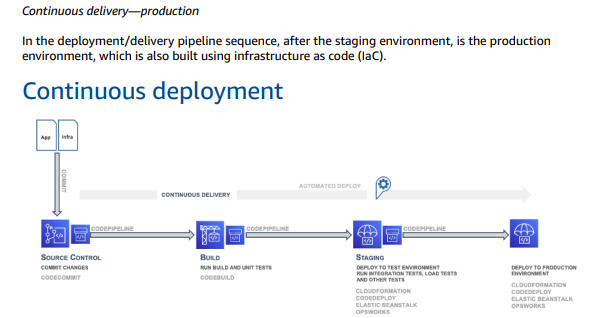
****

The first phase in the CI/CD journey is to develop maturity in continuous integration. You should make sure that all of the developers regularly commit their code to a central repository (such as one hosted in CodeCommit or GitHub) and merge all changes to a release branch for the application. No developer should be holding code in isolation. If a feature branch is needed for a certain period of time, it should be kept up to date by merging from upstream as often as possible. Frequent commits and merges with complete units of work are recommended for the team to develop discipline and are encouraged by the process. A developer who merges code early and often, will likely have fewer integration issues down the road.

You should also encourage developers to create unit tests as early as possible for their applications and to run these tests before pushing the code to the central repository. Errors caught early in the software development process are the cheapest and easiest to fix.

When the code is pushed to a branch in a source code repository, a workflow engine monitoring that branch will send a command to a builder tool to build the code and run the unit tests in a controlled environment. The build process should be sized appropriately to handle all activities, including pushes and tests that might happen during the commit stage, for fast feedback. Other quality checks, such as unit test coverage, style check, and static analysis, can happen at this stage as well. Finally, the builder tool creates one or more binary builds and other artifacts, like images, stylesheets, and documents for the application.

**Continuous delivery—staging**

Continuous delivery (CD) is the next phase and entails deploying the application code in a staging environment, which is a replica of the production stack, and running more functional tests. The staging environment could be a static environment premade for testing, or you could provision and configure a dynamic environment with committed infrastructure and configuration code for testing and deploying the application code. **Continuous deployment**

The final phase in the CI/CD deployment pipeline is continuous deployment, which may include full automation of the entire software release process including deployment to the production environment. In a fully mature CI/CD environment, the path to the production environment is fully automated, which allows code to be deployed with high confidence.

**Exercise 4:**

**Configure the web application and Version control using Git using Git commands and version control operations**

**Create the web application repository, add all the team members & enforce some branch and review protections.**

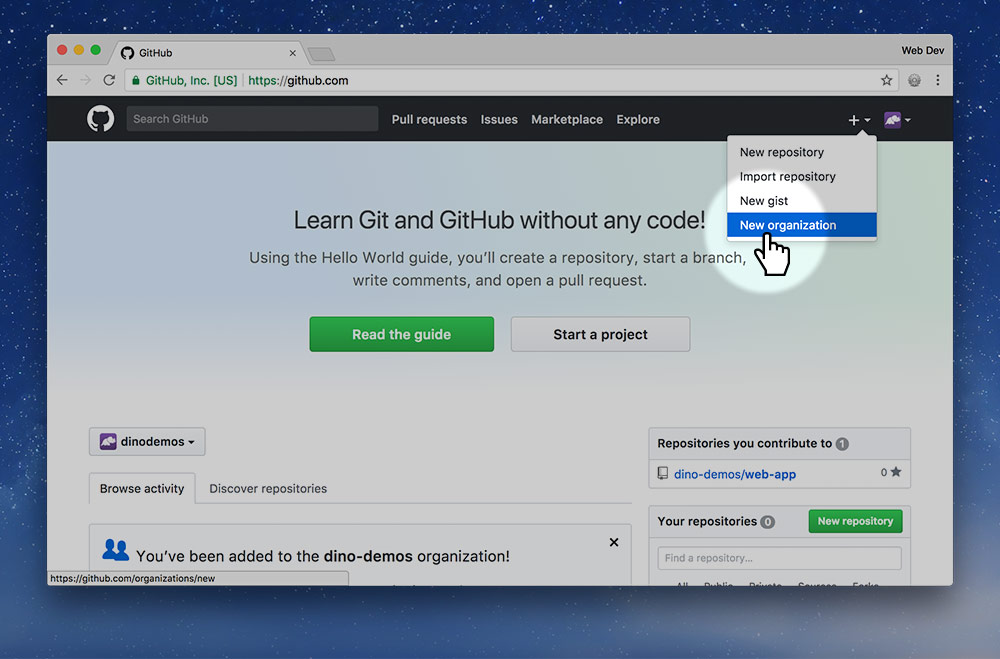
## Goal

## We’re going to start the basic repository, add all our teammates & set up a better GitHub code review process to enforce code reviews for each other.

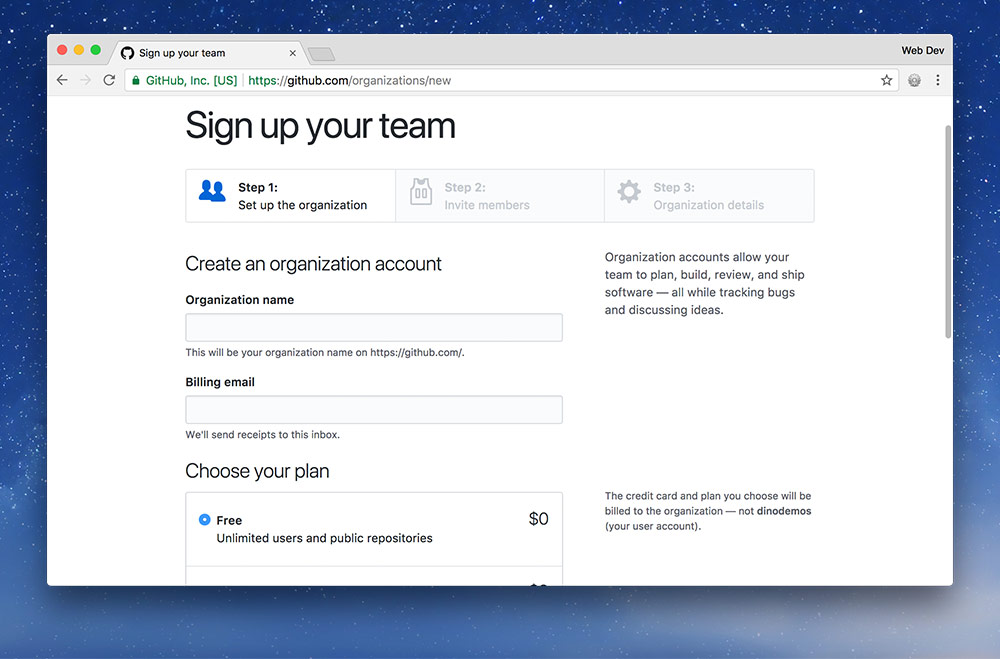
## 1.Create an organization

For this project you’ll all be owners of the code-base and the application. To better support this we’re going to create an organization, that way the repository won’t live directly in a single user’s GitHub account.

The project manager should go to GitHub and create a new organization.

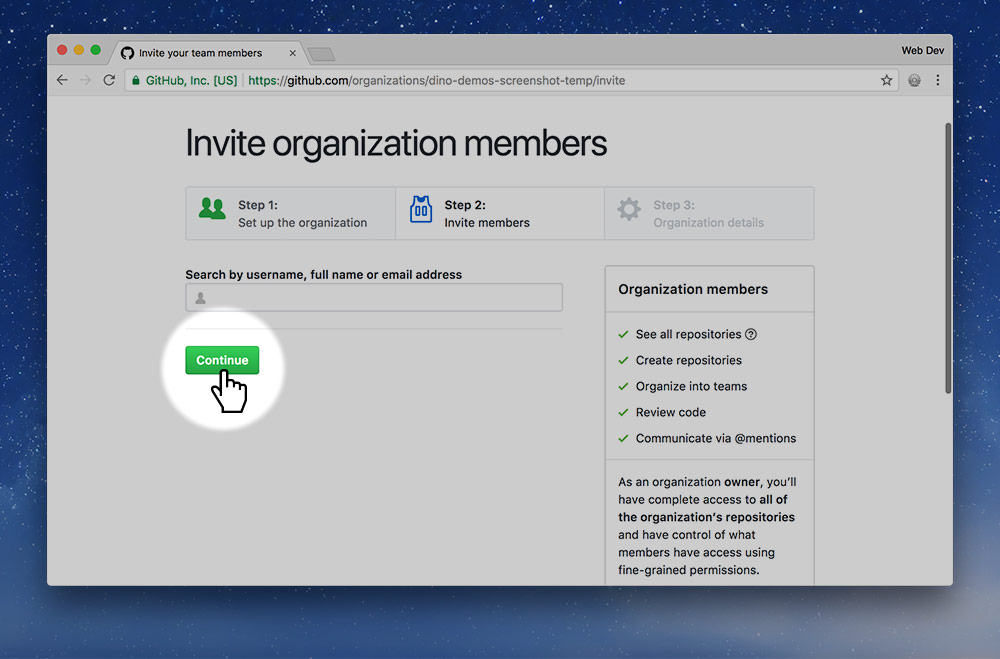


In the + menu go to “New organization”.

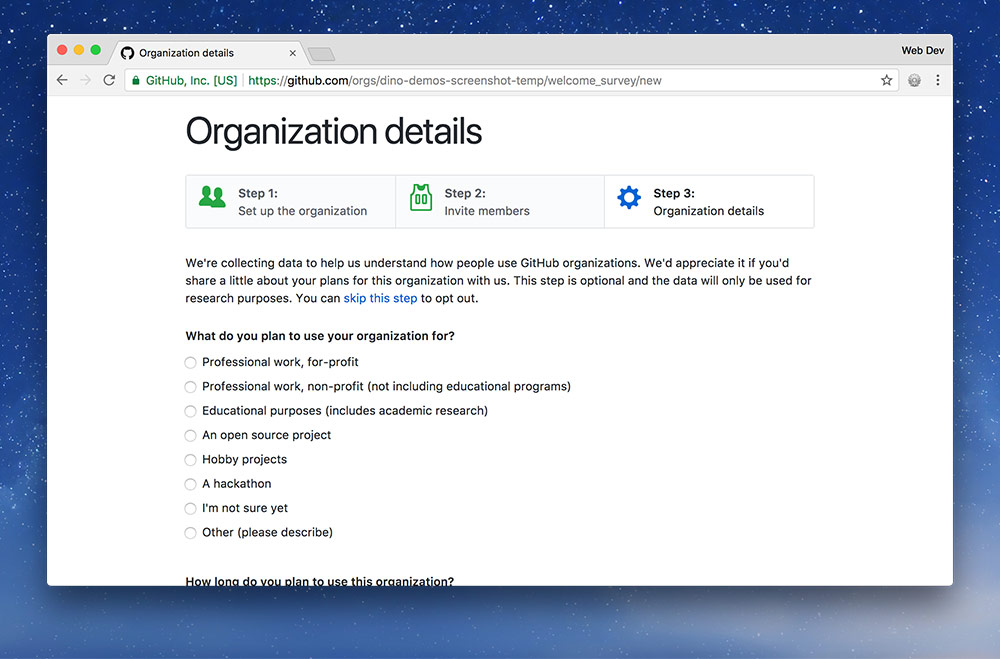


Fill in following pieces of information about your organization:

1. The organization URL, will end up like this: https://github.com/your-org-name
2. The email address of your project manager.
3. Select the “Free” organization type.
4. Then press the green “Create organization” button.



Skip the “Invite organization members” step and continue to the survey.



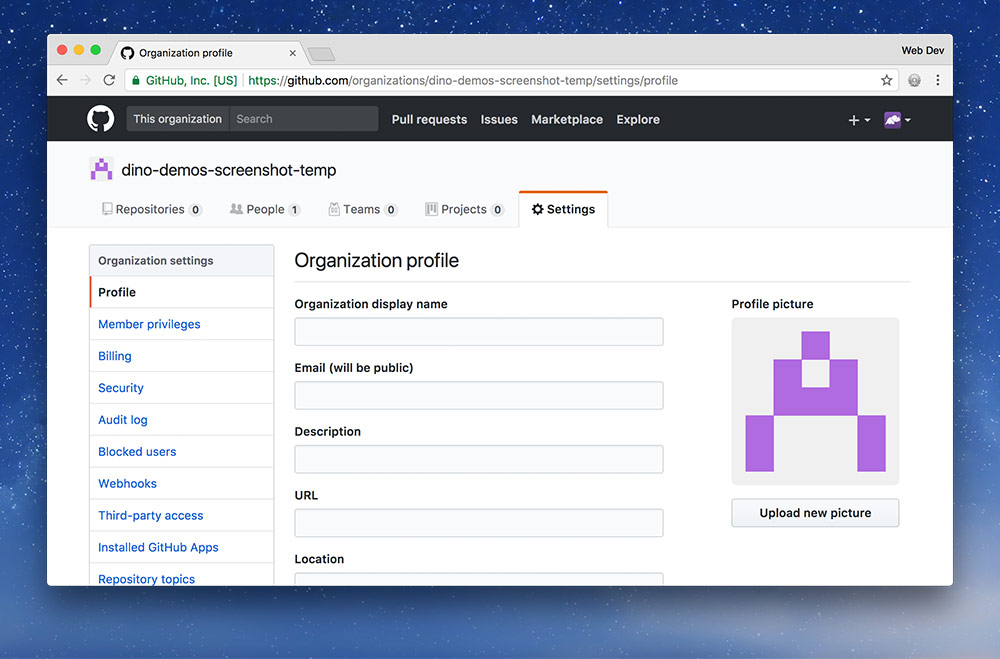
On the survey, use the following answers to the three questions:

1. “Educational purposes (includes academic research)”
2. “A few weeks to months”
3. “5 or fewer”

Then press “Submit”.

### Customize the organization

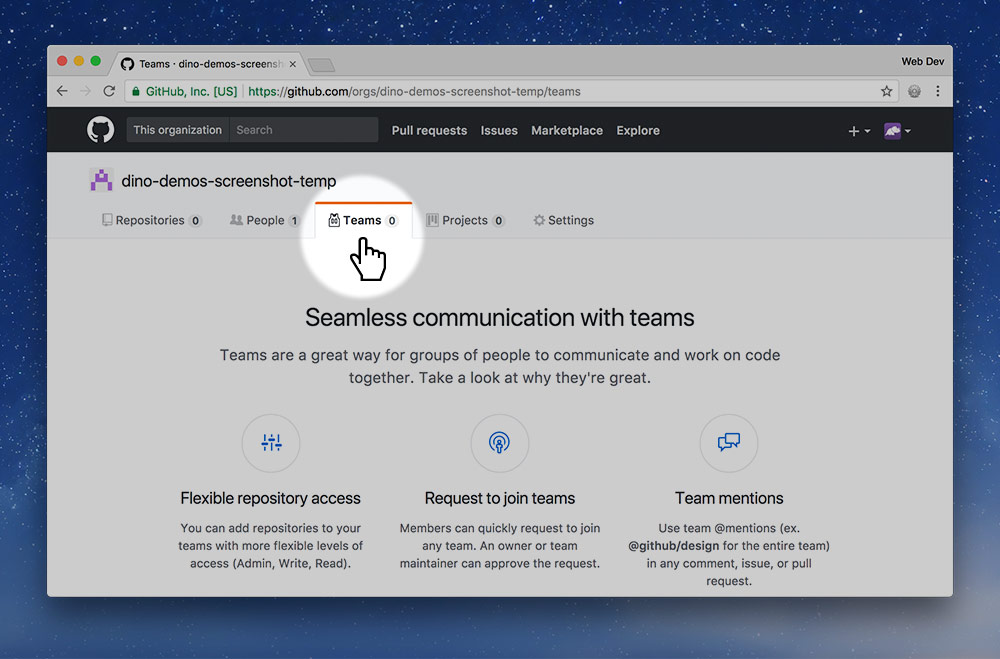
If you’d like you can also customize some properties of the organization by going to the “Settings” tab.



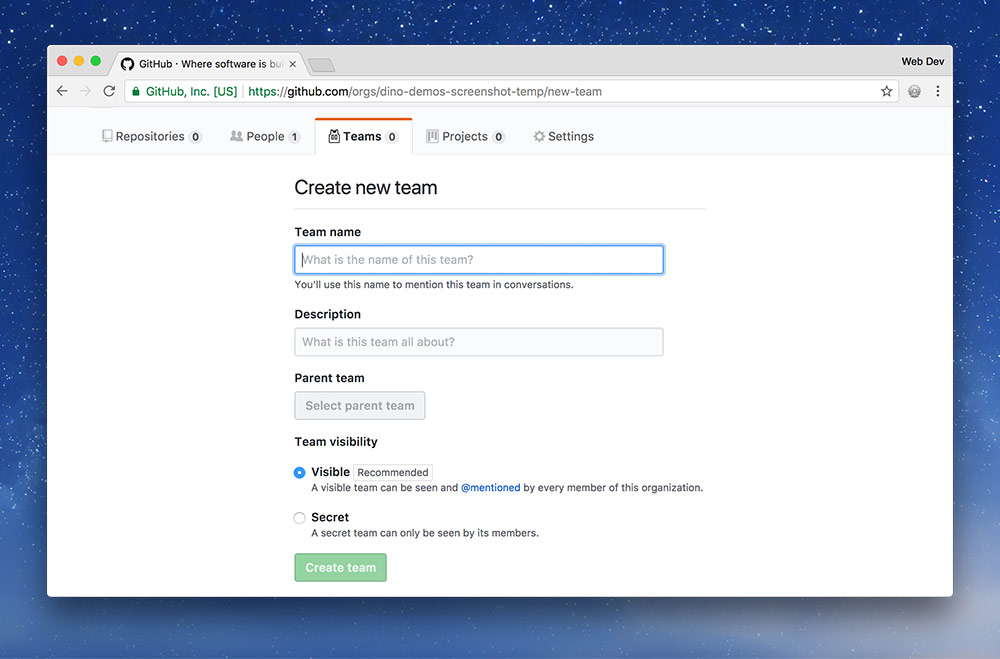
Upload a profile picture of your organization and enter a better “Organization display name”. Fill out any other details you’d like.

## 2.Add all the team members

Next up we’re going to create a team to give everybody equal permissions to the organization and its repositories.



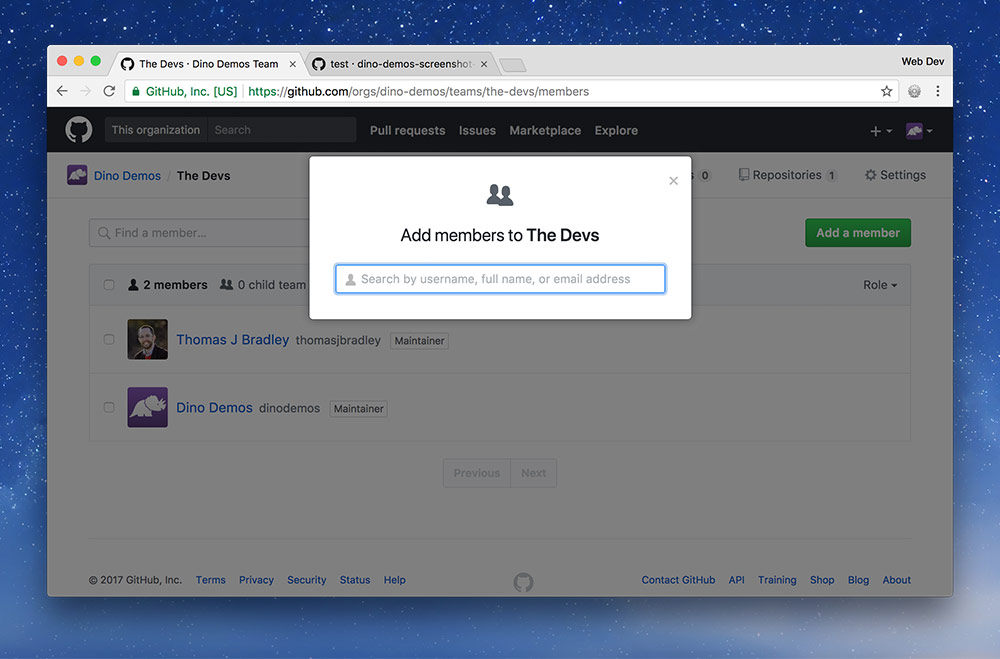
Click the “Teams” tab and press “New Team”.



Fill in all the details about your team: the name you came up with, a description of your awesomeness, then press “Create team”



Next up we need to add members to the team because the project manager is the only member right now. So go to the “Members” link.



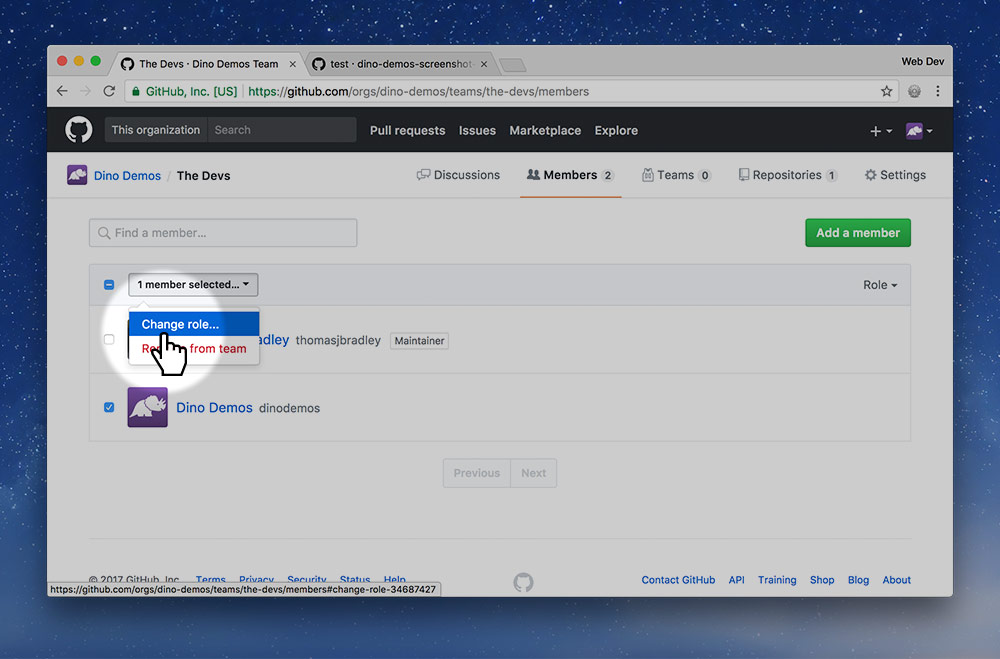
**For each person in your group press “Add member” and type in their GitHub username.** It will autocomplete.

## 3.Approve being a team member

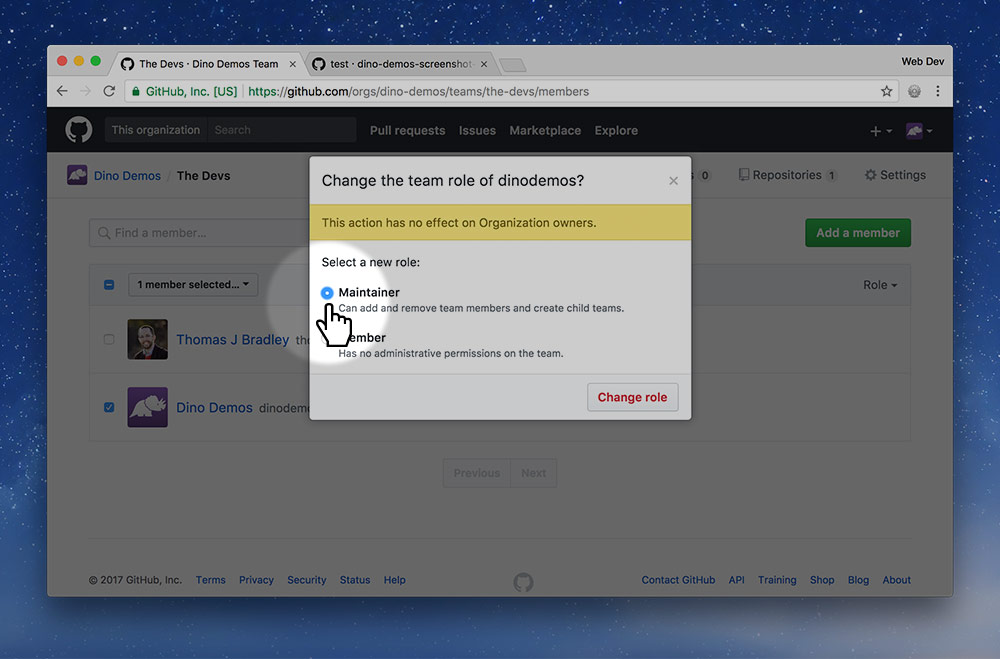
Now every person in the group will get an email from GitHub. **You’ll each have to approve being added to the team before continuing to the next step.**

## 4.Make everyone a maintainer

Right now the project manager is the most important member of the team but we want everybody to have equal importance. So, we’re going to change everybody’s role to be “Maintainer”.



Select all the team members except the PM and click the “№ members selected” button. Then press “Change role”.



Change everybody to the “Maintainer” role and save the changes.

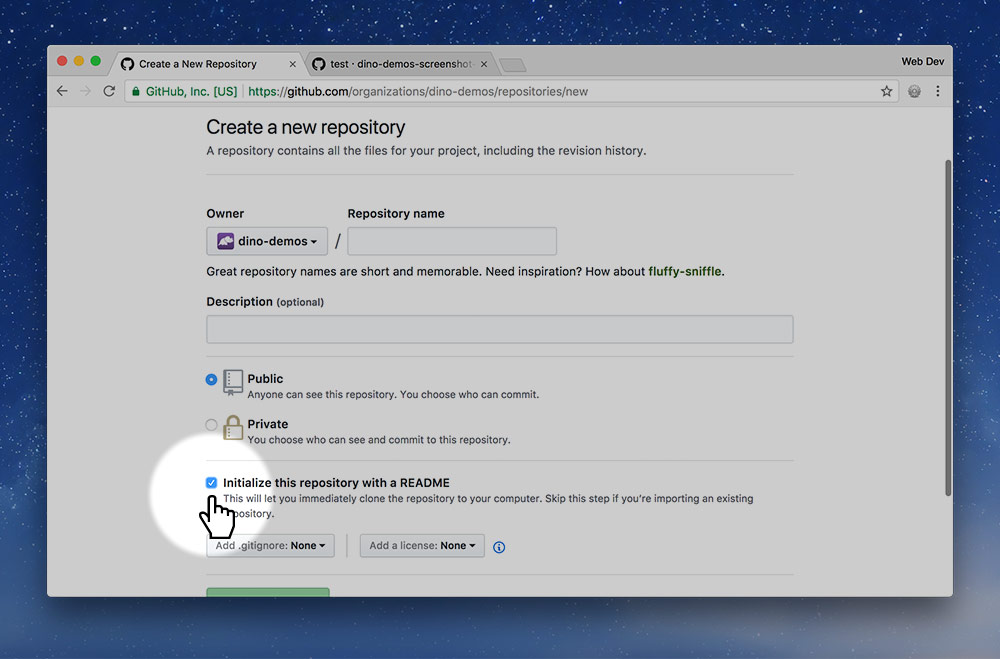
Now everybody can contribute to the project on the same level including configuring settings and managing teams.

## 5.Create the repo in the org

The whole point of this is to make a shared repository to write code in—that’s what we’re going to do next.



So, go back to the organization’s start page, to the “Repositories” tab and press the green “New” button.



Fill in the standard details for your repository, you could name the repo something generic, like web-app, because you haven’t decided what you’re actually making yet.

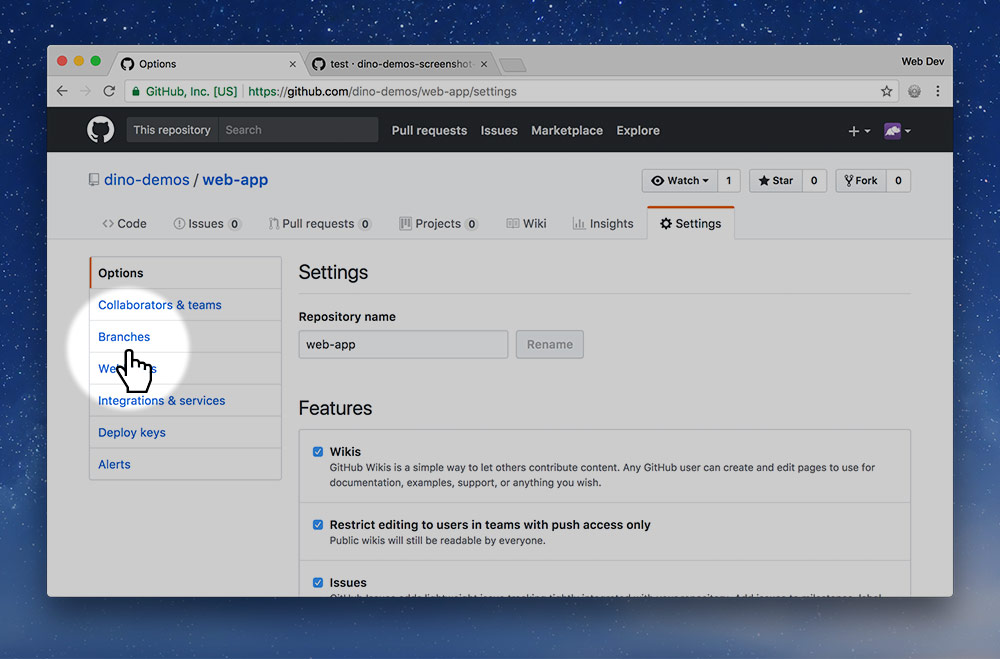
**Make sure to enable “Initialize this repository with a README!”**

## 6.Protect the master branch

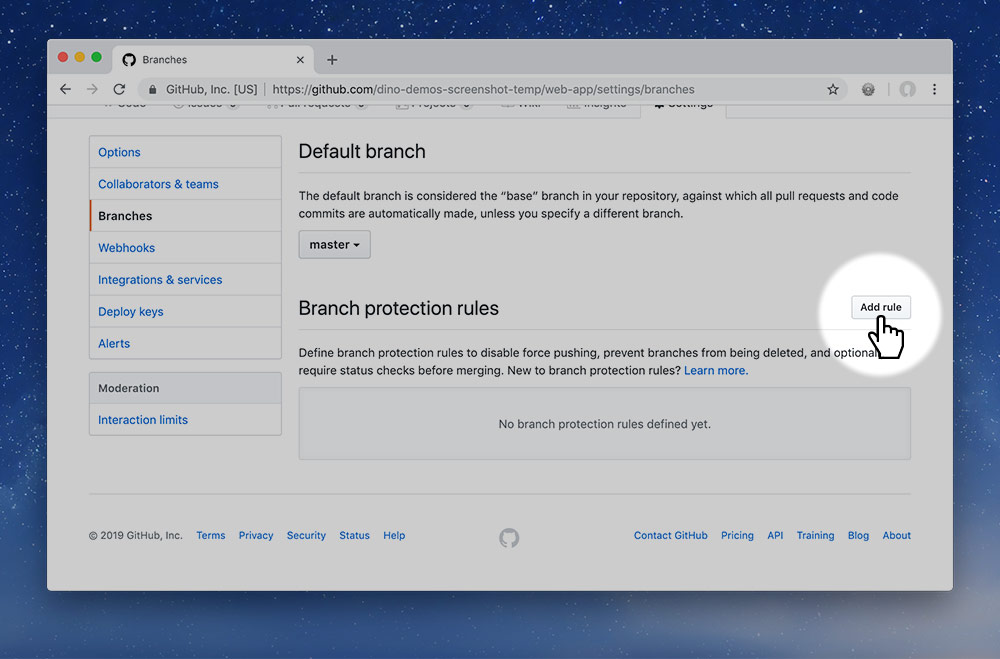
As part of good software development practices we want to isolate the development code from the good, live code. Our master branch should always be in a good state so that it can be live on the website.

We do all our work on other branches and merge those into master only when the code is good enough. This process is a little different from what we’ve been following before.

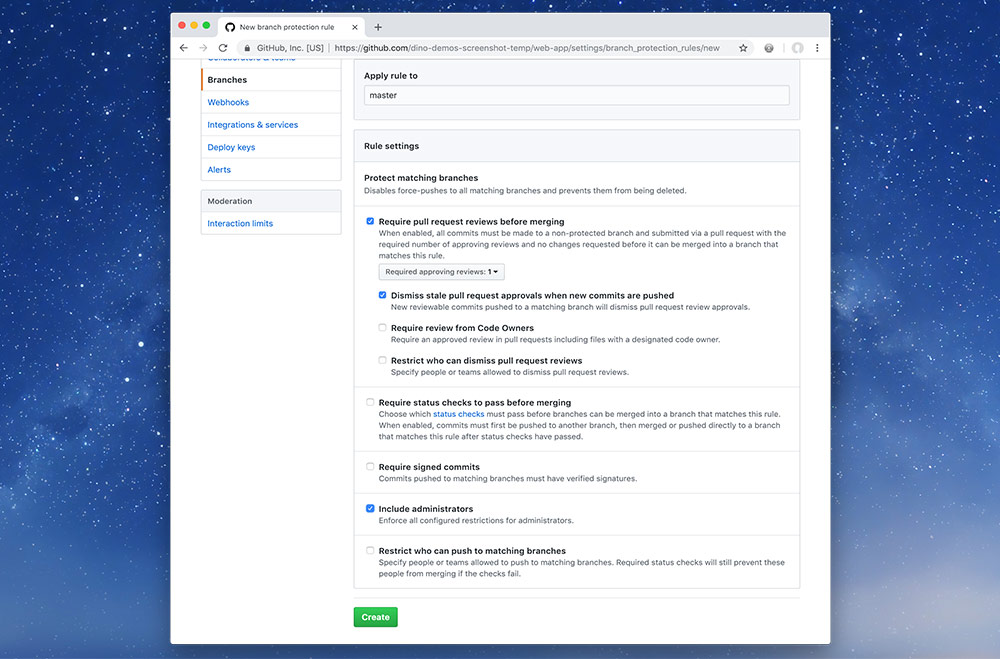
So we’re going to protect the master branch so we can’t accidentally commit or push into that branch. We’re going to enforce a system of code reviews and approvals to help protect the master branch from getting ugly.



In the repo’s “Setting”s tab click the “Branches” button in the sub-navigation.



A little bit down the page is a “Branch protection rules” repository. Press the “Add rule” button.



Finally we need to configure how the master branch will be protected from our messing about.

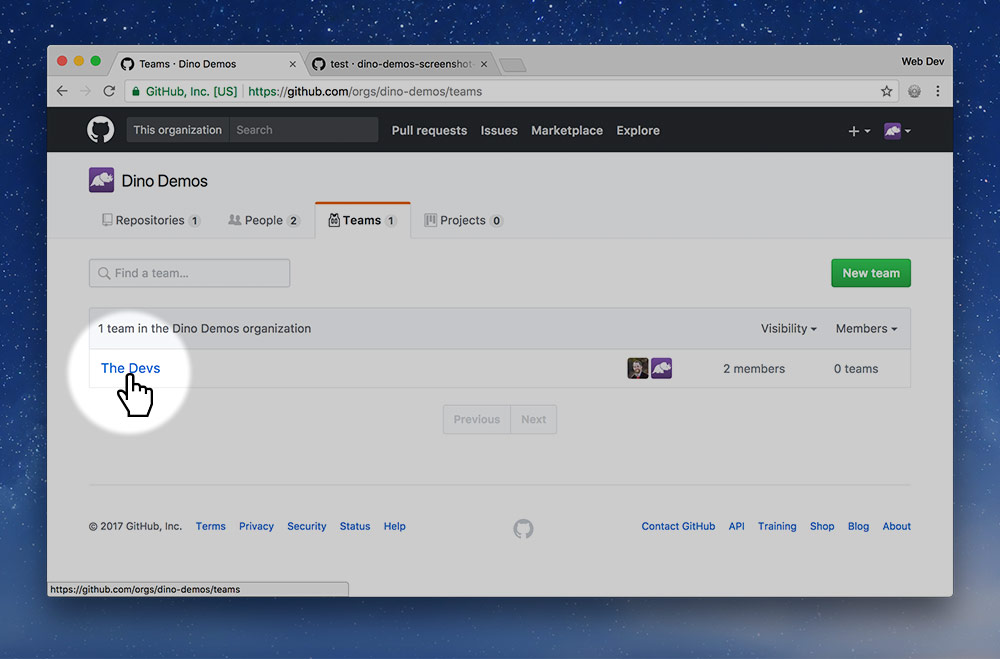
Select the following options on the screen and save:

1. Type master into the “Apply rule to” input field  
   This will prevent you from committing and pushing to the master branch—you’ll have to create a different branch.
2. “Require pull request reviews before merging”  
   This will require everybody to get someone else to approve their code changes before they can be implemented into the master branch.
3. “Dismiss stale pull request approvals when new commits are pushed”  
   If someone has approved the code changes, but then the original author creates a new commit, the approval will be voided.
4. “Include administrators”  
   Enforces these rules on everybody, even if they are an admin of the project (which you all are).

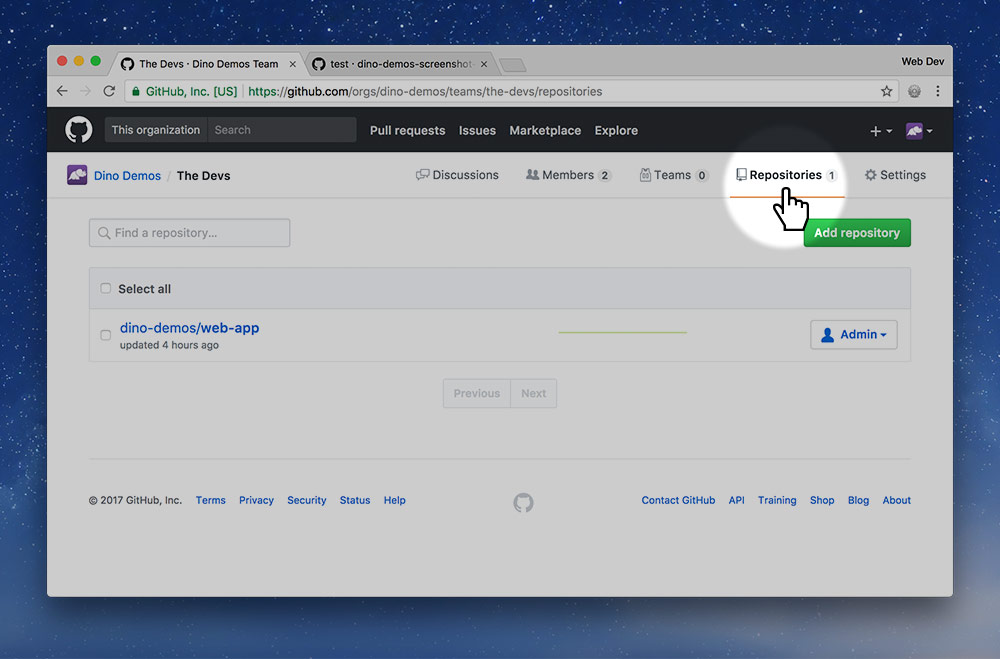
**Make sure to save the changes.**

## 7.Make the team members admins

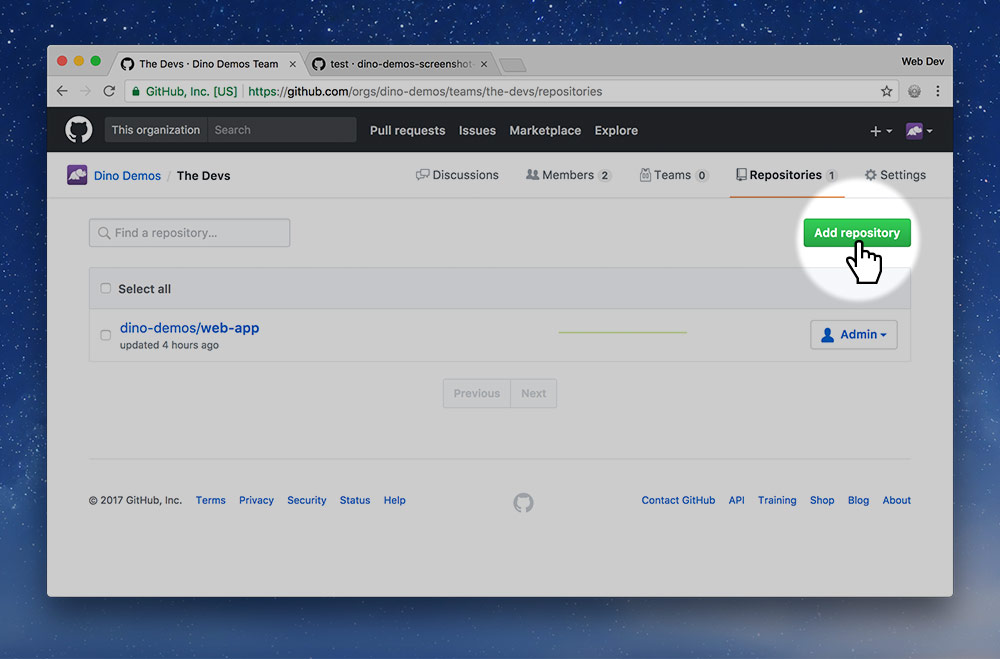
Next up we’re going to give everybody administrator access to this repository. This will allow everybody access to configure and adjust the repository. Our goal is to make everybody equal partners in the project.



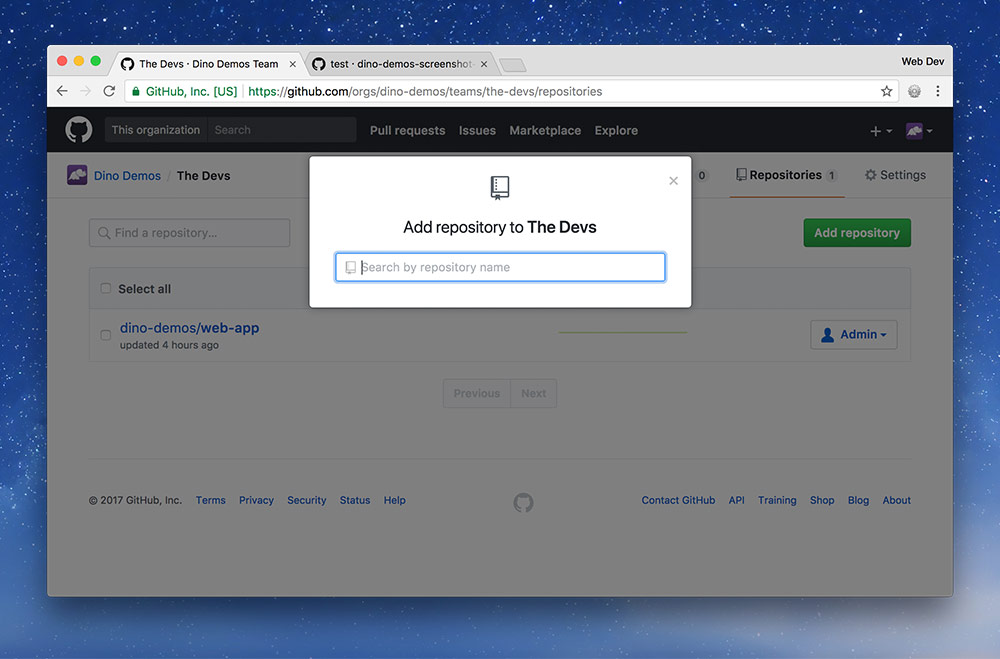
Go back to the organization page and select the “Teams” tab, then click on your team’s name.



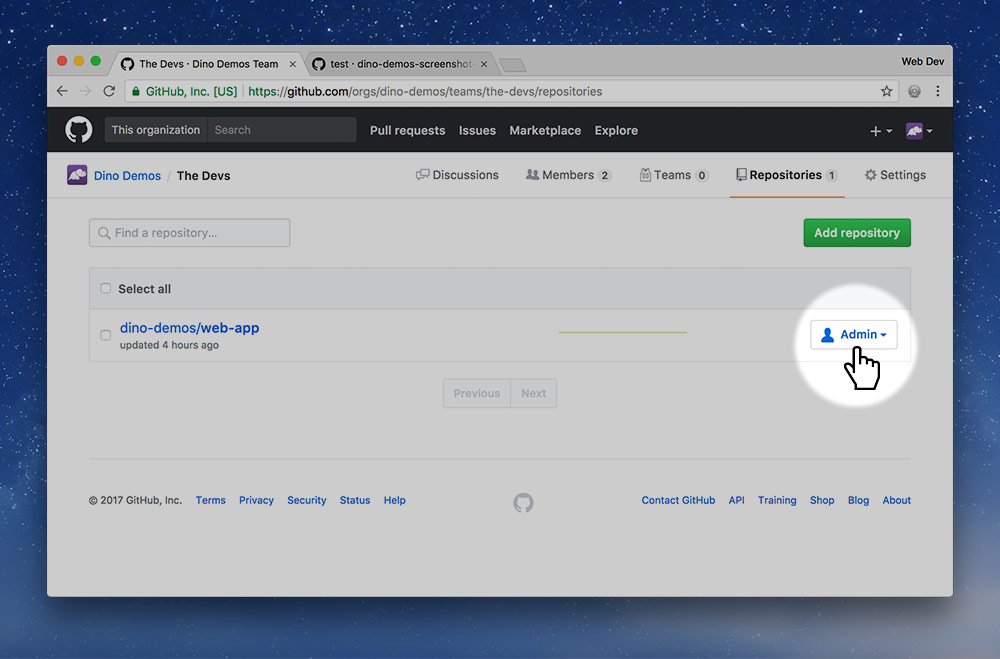
Go to the “Repositories” tab for your team.



Press the big green “Add repository” button.



Search for the name of your repository and add it to the list.



Double check that your team is set as “Admin” for the repository—which I think is the default.

**Now our organization and team is ready to go!**

## 8.Clone to everybody’s computer

Finally, something for all the team members to do, though it’s really small at this point: clone the repo to your computer.

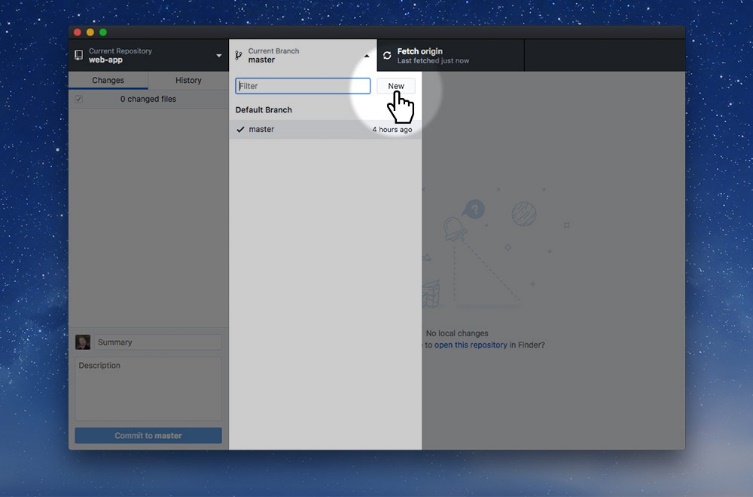
Everybody should go to the web-app repository and clone it to their computer using the standard procedure.

## 9.Create a temporary branch

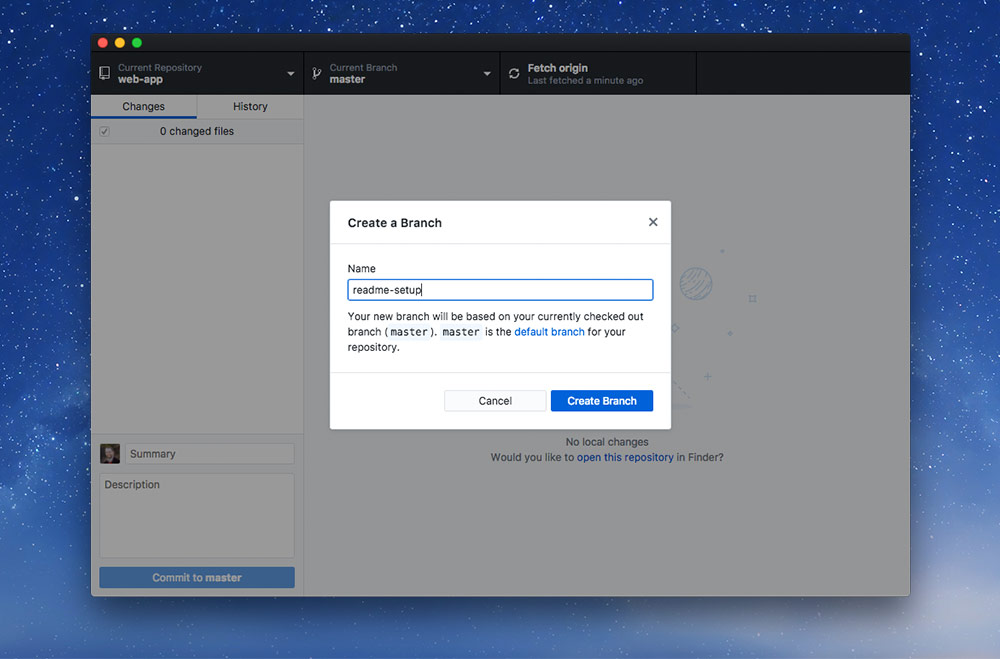
This step is for the dev lead to complete. We’re going to create the readme with some basic information for now that can be filled in with more detail later.

After cloning the repository we’re going to make a new branch in the GitHub Desktop app.

Remember that we cannot commit and push to the *master* branch.



Click the branches button and press the “New” button.



Name the new branch something like setup-readme then press “Create branch”. This is the process you’ll go through every time you code a new feature—we’ll discuss this in more detail later.

## 10.Add a descriptive readme

Open up the folder in your code editor and edit the README.md file, adding a few details. Something like this will work:

**# Web app prototype**

This repository is a prototype for a web app created in Graphic Design’s Web Dev 6 course.

**## Team members**

- Thomas J Bradley <https://github.com/thomasjbradley>

- Dino Demos <https://github.com/dinodemos>

**Save and commit that.** Make sure you’re not committing to the master branch.

### Commit conventions

*Don’t forget to follow the*[*Git commit best practices*](https://learntheweb.courses/topics/commit-message-cheat-sheet)*.*

## 10.Add a GitIgnore file

Create a new file in the repository named .gitignore and add this information to it. **Copying is enabled for efficiency.**

# Jekyll

\_site

.jekyll-metadata

.jekyll-cache

# Adobe files

\*.pdf

\*.ai

\*.psd

\*.indd

\*.indb

# Compressed files

\*.zip

\*.gz

\*.tar

\*.7z

# Type faces

\*.otf

\*.ttf

\*.woff

\*.eot

\*.ttc

# Video & audio

\*.mov

\*.mp4

\*.m4v

\*.f4v

\*.f4p

\*.ogv

\*.webm

\*.flv

\*.mp3

\*.m4a

\*.f4a

\*.f4b

\*.oga

\*.ogg

\*.opus

# Folder view configuration files

.DS\_Store

Desktop.ini

# Thumbnail cache files

.\_\*

Thumbs.db

# Files that might appear on external disks

.Spotlight-V100

.Trashes

**Save and commit that.**

## 12.Add an EditorConfig file

Create a new file named .editorconfig in the folder and add this information to it. **Copying is enabled for efficiency.**

; editorconfig.org

root = true

[\*]

charset = utf-8

indent\_style = space

indent\_size = 2

trim\_trailing\_whitespace = true

end\_of\_line = lf

insert\_final\_newline = true

**Save and commit that.**

## 13.Add a GitAttributes file

You only need perform this step if one or more of your team members use a Windows computer—but it’s good practice to do it anyways.

Create another new file named .gitattributes and add this information to it. **Copying is enabled for efficiency.**

# Force Unix LF to make code consistent across platforms

# Helps Markbot cheat detection to work properly

\*.html text eol=lf

\*.css text eol=lf

\*.js text eol=lf

\*.md text eol=lf

\*.yml text eol=lf

\*.txt text eol=lf

\*.svg text eol=lf

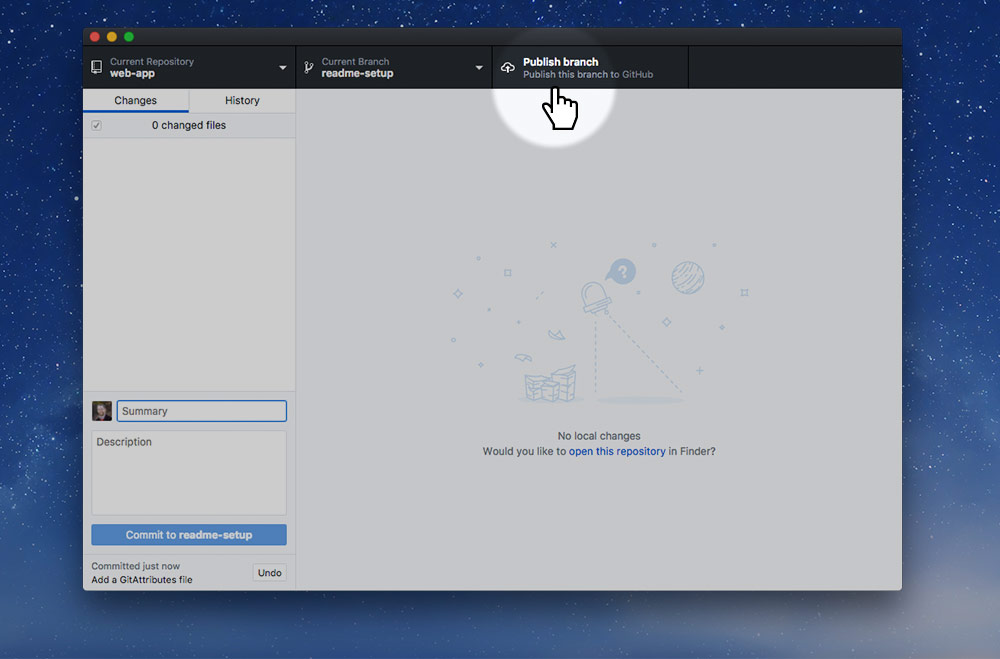
\*.lock text eol=lf

.editorconfig text eol=lf

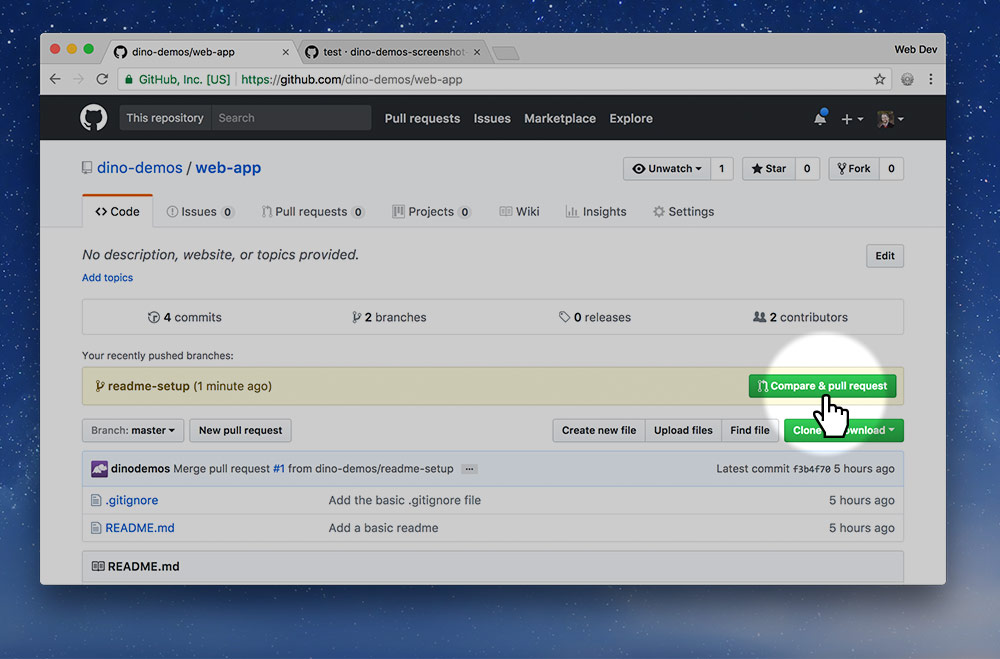
**Save and commit that.**

## 14.Create a pull request

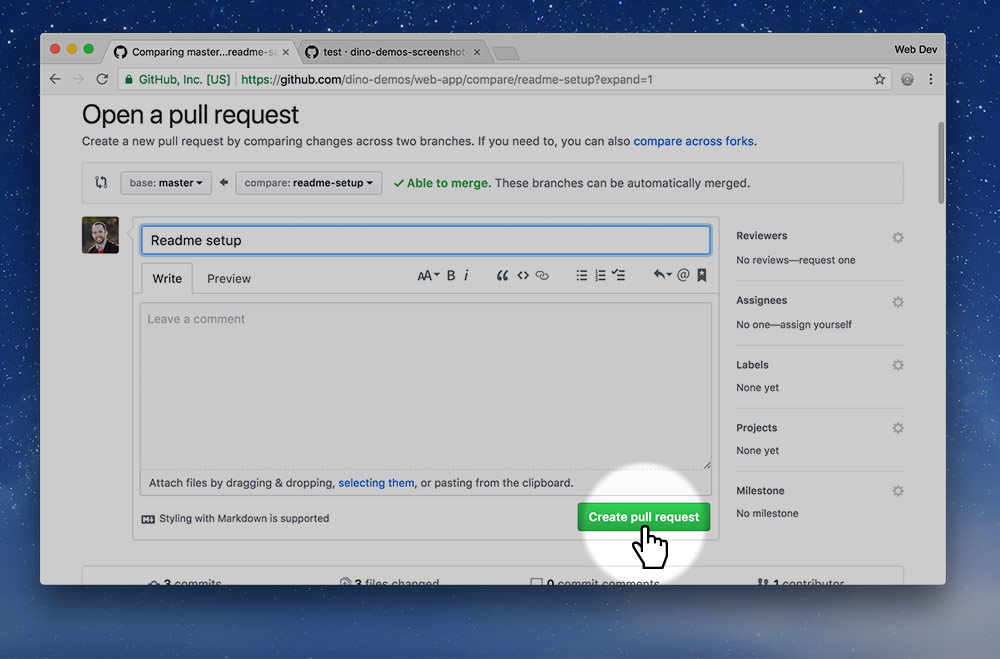
Our next goal is to get this code into the master branch so it’s available on everybody’s computer. Step one is to publish our new branch to GitHub.



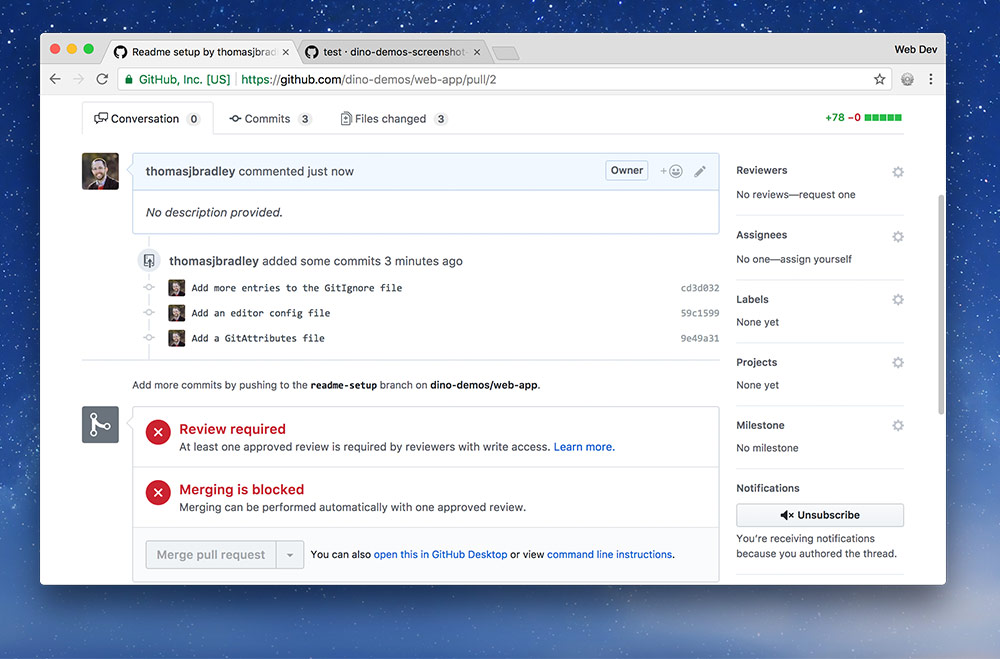
Press the “Publish branch” button which will send your setup-readme branch to GitHub.



If you go to the repo’s page you’ll see the yellow new branch alert. Press the “Compare & pull request” button.



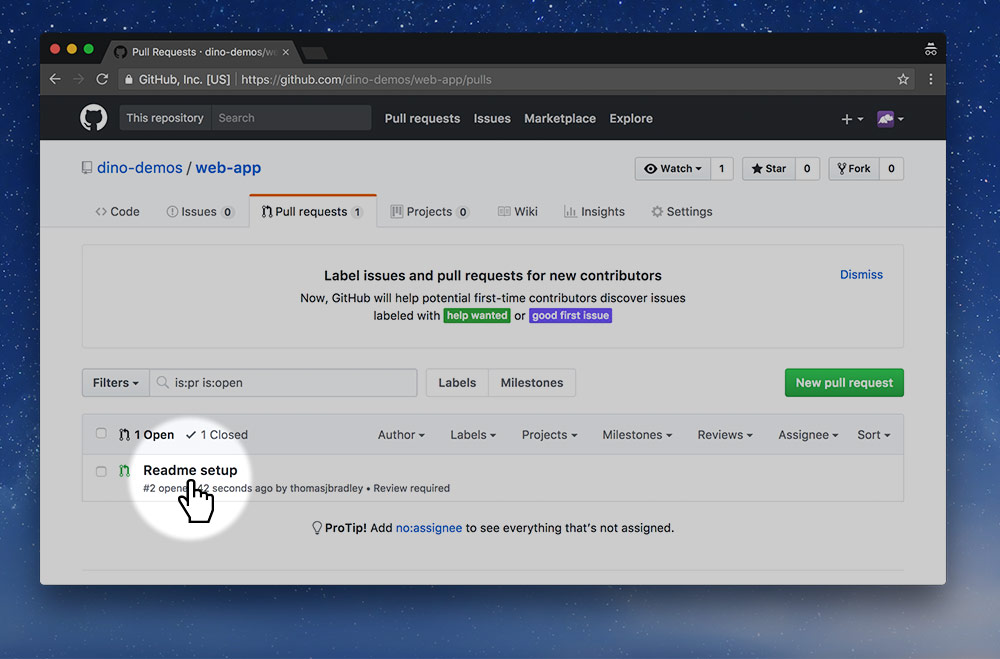
Describe what changes you made in the branch you just published and press the “Create pull request” button.



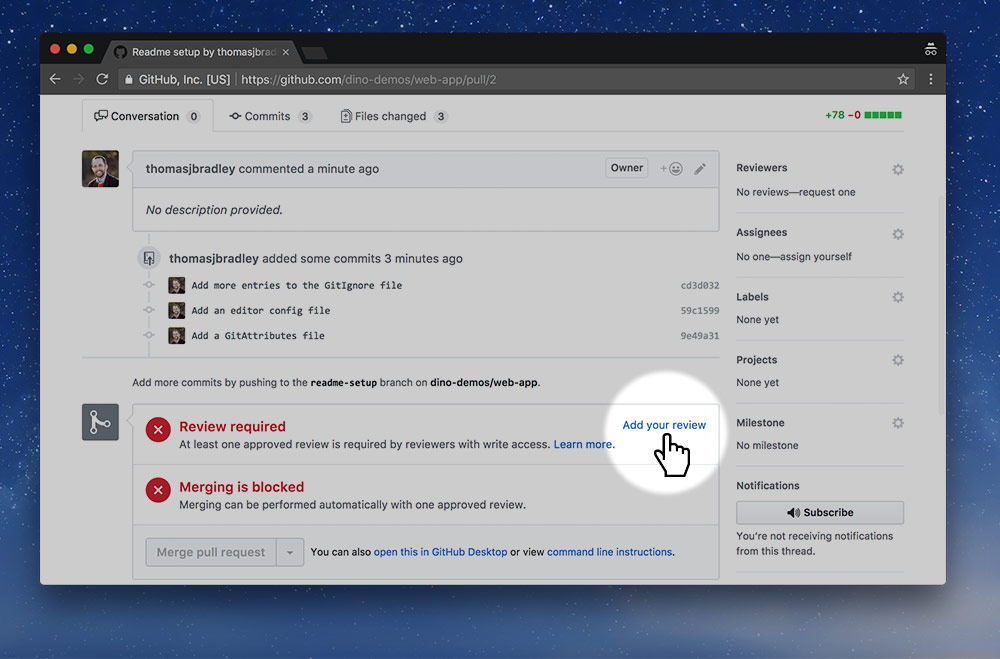
You’ll notice that next screen says a code review is required. So, now we hand the process off to another teammate, the UX lead.

## 15.Approve & merge the request

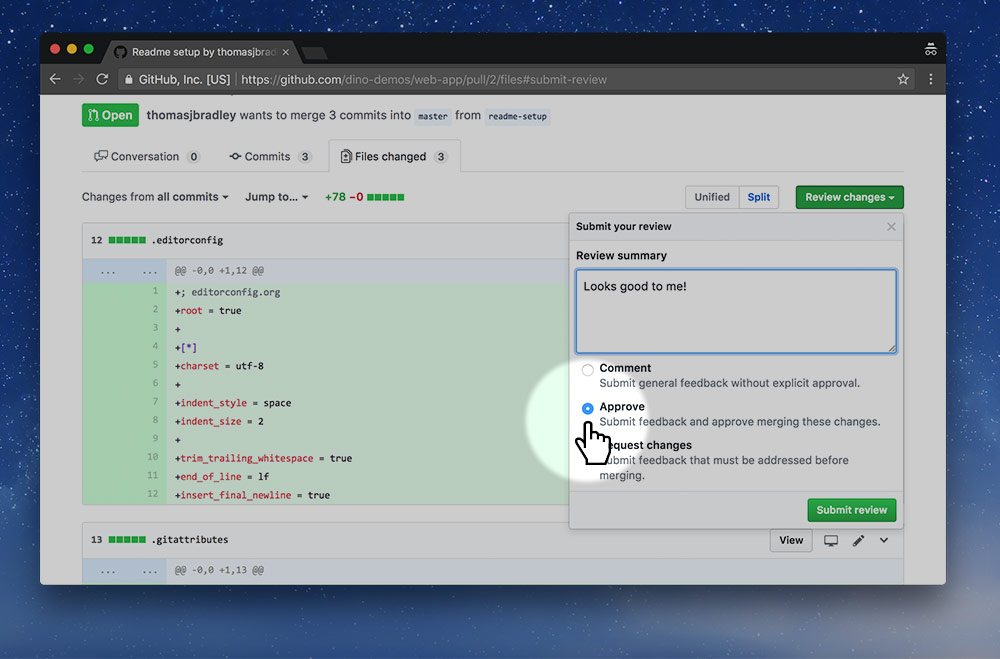
This task is for the UX lead. Go to the GitHub repo page and press the “Pull requests” tab.



On the “Pull requests” page you should see the new pull request, click into it.

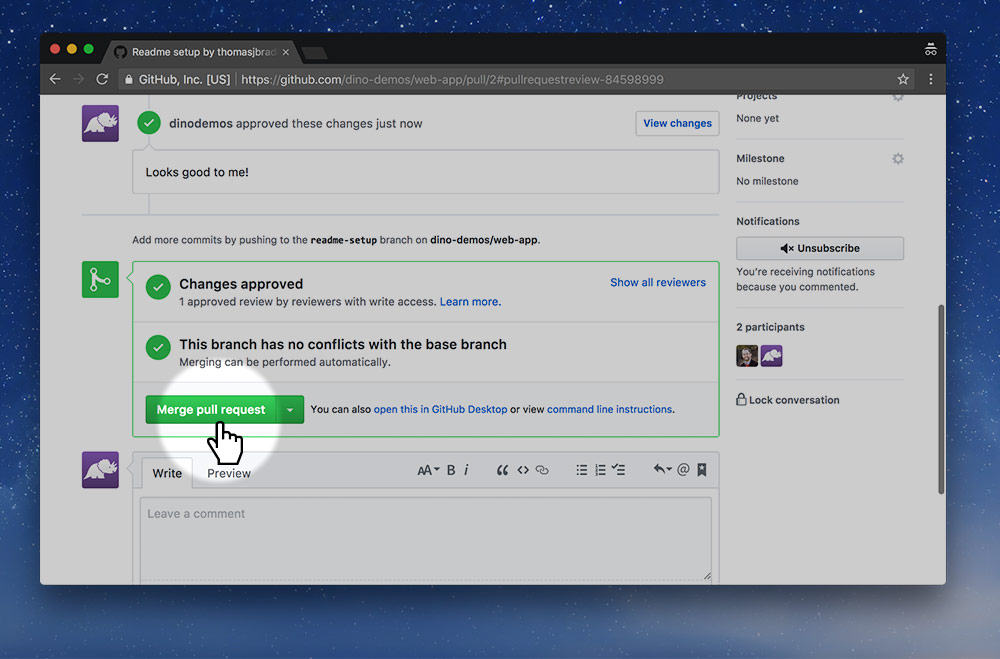


You’ll notice that the PR requires a code review to move forward. It’s up to do you do that review. Press the “Add your review” link.

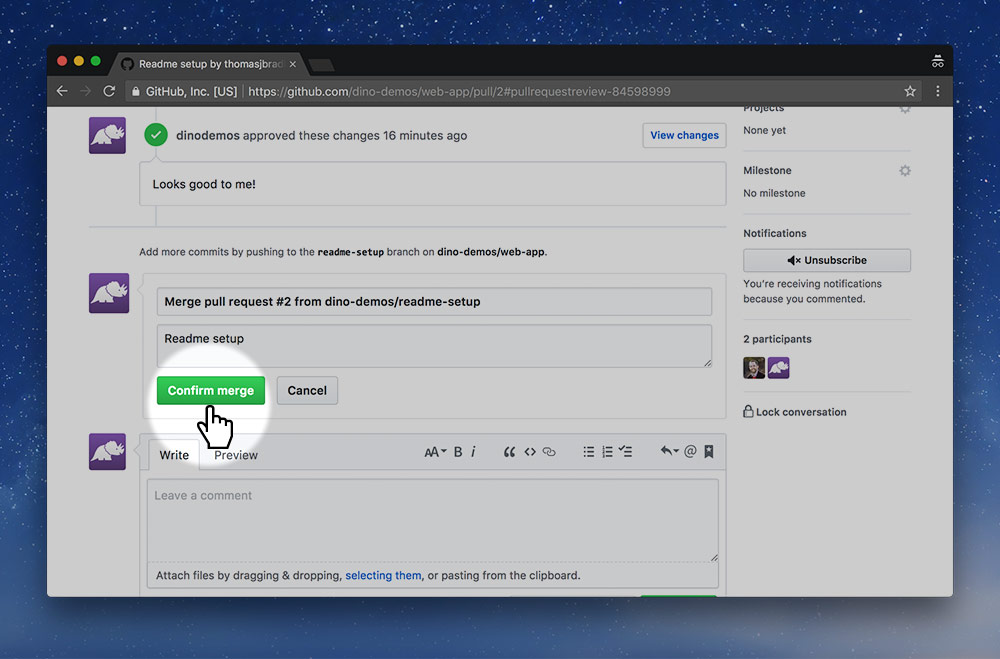


On the next screen you’ll see all the code contributions that were made. Double-check there’s no spelling mistakes or problems.

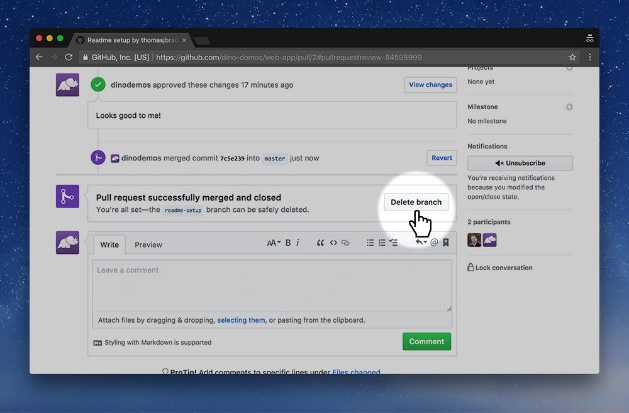
Only if everything is okay, write a small comment, select “Approve”, and press the “Submit review” button.



Since the code has been approved, we’re now ready to merge those changes into the master branch. Press the “Merge pull request” button.



Then press “Confirm merge”

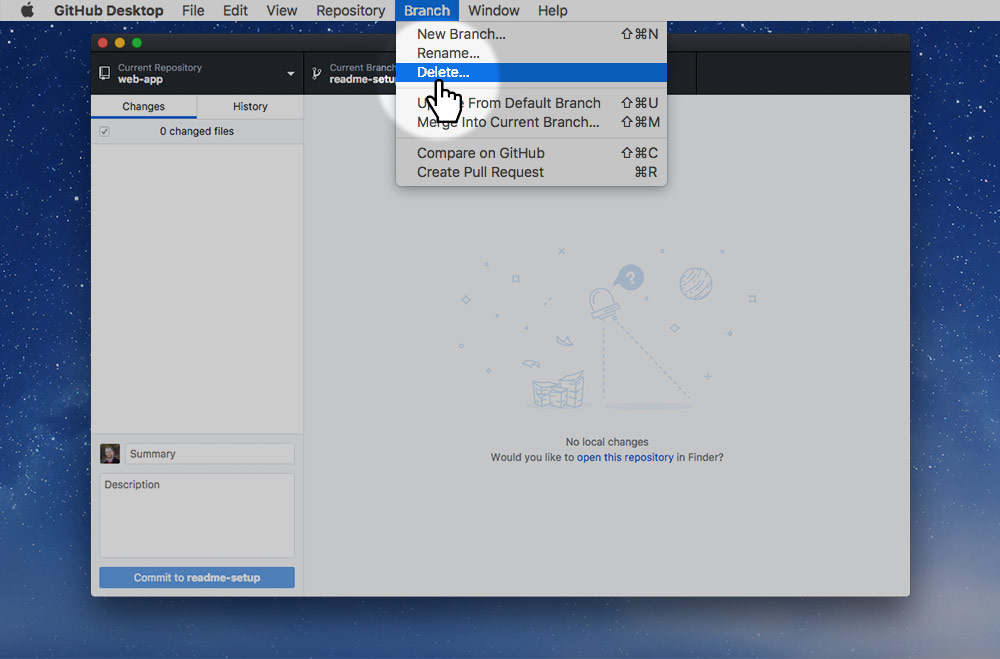


Now that the code is merged, we no longer need the branch so press the “Delete branch” button.

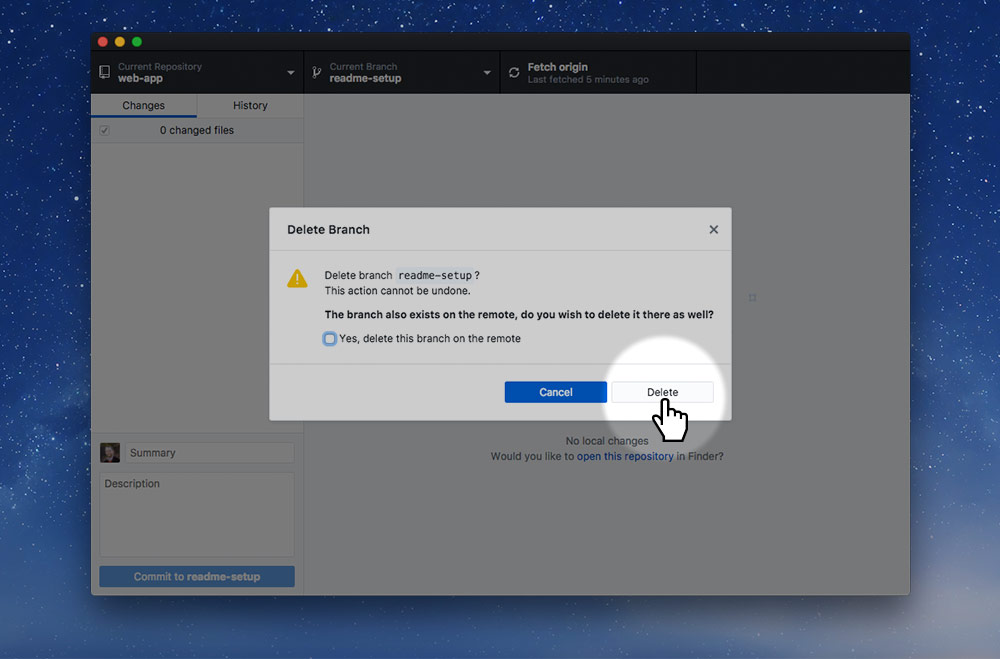
That only deletes it from GitHub.com, not from our local computer. We’re going to do that next.

## 16.Delete the local branch

Back to the dev lead. Now that the code has been merged into the master branch the local setup-readme branch is no longer necessary so we need to delete it.



Inside the GitHub Desktop app, in the menu, go to: Branch > Delete… to remove the local branch.



Confirm the deletion by pressing the “Delete” button.

## 17.Pull the changes from GitHub

For everybody to complete. We want to make sure everybody has the most up-to-date version of the code on their computers, so they need to pull the most recent changes from GitHub’s website.

On your own computer go into the GitHub Desktop app and press the “Fetch origin” button. If it switches and says “Pull” do that too.

**Double check that your local copy of the website has the correct README.md, .editorconfig, .gitignore and .gitattributes files.**